Titre: Title:	Analyse des performances de stockage, en mémoire et sur les périphériques d'entrée/sortie, à partir d'une trace d'exécution
Auteur: Author:	Houssem Daoud
Date:	2019
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Daoud, H. (2019). Analyse des performances de stockage, en mémoire et sur les périphériques d'entrée/sortie, à partir d'une trace d'exécution [Thèse de doctorat, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/3847/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/3847/
Directeurs de recherche: Advisors:	Michel Dagenais
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

ANALYSE DES PERFORMANCES DE STOCKAGE, EN MÉMOIRE ET SUR LES PÉRIPHÉRIQUES D'ENTRÉE/SORTIE, À PARTIR D'UNE TRACE D'EXÉCUTION

HOUSSEM DAOUD
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
AVRIL 2019

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

ANALYSE DES PERFORMANCES DE STOCKAGE, EN MÉMOIRE ET SUR LES PÉRIPHÉRIQUES D'ENTRÉE/SORTIE, À PARTIR D'UNE TRACE D'EXÉCUTION

présentée par : DAOUD Houssem
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. OZELL Benoit, Ph. D., président
M. DAGENAIS Michel, Ph. D., membre et directeur de recherche
M. QUINTERO Alejandro, Doctorat, membre
M. MCGUFFIN Michael J., Ph. D., membre externe

DÉDICACE

Je dédie cette thèse à ma mère Amel,
source inépuisable de tendresse, de patience et d'amour...

REMERCIEMENTS

Je remercie tout d'abord mon directeur de recherche, Michel Dagenais, d'avoir cru en moi. La confiance et l'autonomie qu'il m'a accordées, son soutien constant et la qualité de son suivi ont été essentiels à la réussite de ce projet.

Un grand merci à Naser pour ses conseils qui ont été particulièrement précieux. Son expérience et sa sagesse ont été une grande source d'inspiration pour moi. Je tiens aussi à remercier Geneviève de son aide tout au long du projet.

Je remercie également tous les membres du laboratoire de recherche DORSAL, et en particulier Hani, Adel, Majid, Anas, Iman, Ahmad, Vahid, Marie, Pierre, Francis, Suchakra, Thomas, Mohamad et Julien, pour leur bonne humeur et leur soutien. Je remercie aussi les partenaires industriels, en particulier Francois, Jim, Octavian et Matthew.

Je remercie mes amis Bechir, Wael, Sofiene, Mohamed Ali, Rabeh, Elyes, Adnan, Safouen, Nader, Malek, Yassine, Omar, Ghassen et Fraj, pour les bons moments passés ensemble. Je remercie aussi Hichem et Sab, avec qui j'ai appris à apprécier le moment présent. Un grand merci à Phoenix pour ses encouragements et son soutien. Il a toujours été là pour moi et il m'a appris qu'il faut toujours rêver grand.

Je remercie mon cher frère Zied pour son amour et son soutien inconditionnel. Il est pour moi une source de bonheur, de joie et d'inspiration. Je remercie aussi Hana et le petit Mohamed Aziz. Un merci également à tous les membres de la famille Kaabar.

Finalement, un grand merci à ma mère Amel. Elle est ma source de motivation, et ma raison d'être. Sans elle, je n'aurais jamais pu arriver là où je suis aujourd'hui. Les mots ne peuvent pas exprimer mes sentiments de gratitude et d'amour envers elle...

RÉSUMÉ

Le stockage des données est vital pour l'industrie informatique. Les supports de stockage doivent être rapides et fiables pour répondre aux demandes croissantes des entreprises.

Les technologies de stockage peuvent être classifiées en deux catégories principales : stockage de masse et stockage en mémoire. Le stockage de masse permet de sauvegarder une grande quantité de données à long terme. Les données sont enregistrées localement sur des périphériques d'entrée/sortie, comme les disques durs (HDD) et les Solid-State Drive (SSD), ou en ligne sur des systèmes de stockage distribué. Le stockage en mémoire permet de garder temporairement les données nécessaires pour les programmes en cours d'exécution. La mémoire vive est caractérisée par sa rapidité d'accès, indispensable pour fournir rapidement les données à l'unité de calcul du processeur.

Les systèmes d'exploitation utilisent plusieurs mécanismes pour gérer les périphériques de stockage, par exemple les ordonnanceurs de disque et les allocateurs de mémoire. Le temps de traitement d'une requête de stockage est affecté par l'interaction entre plusieurs sous-systèmes, ce qui complique la tâche de débogage. Les outils existants, comme les outils d'étalonnage, permettent de donner une vague idée sur la performance globale du système, mais ne permettent pas d'identifier précisément les causes d'une mauvaise performance. L'analyse dynamique par trace d'exécution est très utile pour l'étude de performance des systèmes. Le traçage permet de collecter des données précises sur le fonctionnement du système, ce qui permet de détecter des problèmes de performance difficilement identifiables.

L'objectif de cette thèse est de fournir un outil permettant d'analyser les performances de stockage, en mémoire et sur les périphériques d'entrée/sortie, en se basant sur les traces d'exécution. Les défis relevés par cet outil sont : collecter les données nécessaires à l'analyse depuis le noyau et les programmes en mode utilisateur, limiter le surcoût du traçage et la taille des traces générées, synchroniser les différentes traces, fournir des analyses multiniveau couvrant plusieurs aspects de la performance et enfin proposer des abstractions permettant aux utilisateurs de facilement comprendre les traces.

Nous avons conçu et implémenté l'instrumentation nécessaire à l'analyse. Les points de traces utilisés permettent d'avoir une visibilité complète sur le système et de suivre le cycle de vie des requêtes de stockage depuis leur création jusqu'à leur traitement. Nous utilisons le Linux Trace Toolkit Next Generation (LTTng), un traceur libre et à faible surcoût, pour la collecte de données nécessaires à l'analyse. Ce traceur est caractérisé par sa robustesse et sa performance avec les applications hautement parallèles, grâce aux mécanismes de synchronisation

non bloquants utilisés lors de l'enregistrement des événements. Nous avons aussi contribué au développement d'un correctif qui permet à LTTng de récupérer les piles d'appels des événements de trace en mode utilisateur.

Atteindre un équilibre entre la précision de l'analyse et le surcoût du traçage est un défi principal dans notre recherche. Pour chaque analyse proposée, nous effectuons un large éventail de tests pour mesurer le surcoût du traceur en termes d'impact sur le temps d'exécution du programme observé. La taille des traces générées est un autre aspect du surcoût qu'il est important de prendre en considération. Nous proposons deux mécanismes pour éviter d'avoir des traces gigantesques impossibles à analyser. Le premier mécanisme se base sur deux sessions de traçage qui opèrent simultanément. La session légère trace uniquement un nombre réduit d'événements et permet de rapidement détecter des anomalies. En même temps, la session détaillée enregistre la trace complète dans un tampon circulaire gardé en mémoire. Lorsqu'une anomalie est détectée par la session légère, la session détaillée enregistre le contenu du tampon circulaire sur le disque afin d'effectuer une analyse plus avancée. Le deuxième mécanisme proposé utilise un algorithme d'échantillonnage dynamique pour réduire la fréquence de génération d'événements. Au lieu d'enregistrer tous les événements dans la trace, notre algorithme fait des agrégations du côté noyau et génère un événement seulement si une certaine condition est remplie.

Notre outil permet d'analyser des traces collectées à partir de plusieurs machines et sources de données. Les traces sont synchronisées en se basant sur l'horloge monotonique du noyau et sur les relations de causalité sémantique entre les événements dans le cas d'un système distribué. L'analyse des traces se fait en deux phases principales. Lors de la première lecture de la trace, une base de données spéciale est créée pour garder l'historique des états du système au cours du temps. Cette base de données est par la suite utilisée pour fournir les informations nécessaires aux algorithmes d'analyse, ce qui améliore considérablement l'interactivité de notre outil.

Cette méthodologie a été utilisée pour fournir plusieurs types d'analyses. D'abord, nous étudions la performance des périphériques de stockage de masse en nous basant sur les événements noyau collectés depuis le sous-système de stockage de Linux. Ensuite, nous étendons cette analyse pour supporter les systèmes de stockage distribué, notamment Ceph, en utilisant l'instrumentation en mode utilisateur des démons de stockage. Puis, nous fournissons une analyse qui permet de suivre l'utilisation de mémoire par les processus du système. Finalement, nous étudions les méthodes d'allocation dynamique de la mémoire et nous analysons les performances du ramasse-miettes de Java.

Les exemples et les résultats expérimentaux présentés dans cette thèse prouvent l'efficacité

de l'approche. Les analyses sont implémentées en tant que modules d'extension de Trace Compass, un logiciel libre d'analyse de traces. Nous fournissons des métriques de performance et des vues graphiques pour faciliter l'étude du système observé.

ABSTRACT

Data storage is an essential resource for the computer industry. Storage devices must be fast and reliable to meet the growing demands of the data-driven economy. Storage technologies can be classified into two main categories: mass storage and main memory storage. Mass storage can store large amounts of data persistently. Data is saved locally on input/output devices, such as Hard Disk Drives (HDD) and Solid-State Drives (SSD), or remotely on distributed storage systems. Main memory storage temporarily holds the necessary data for running programs. Main memory is characterized by its high access speed, essential to quickly provide data to the Central Processing Unit (CPU).

Operating systems use several mechanisms to manage storage devices, such as disk schedulers and memory allocators. The processing time of a storage request is affected by the interaction between several subsystems, which complicates the debugging task. Existing tools, such as benchmarking tools, provide a general idea of the overall system performance, but do not accurately identify the causes of poor performance. Dynamic analysis through execution tracing is a solution for the detailed runtime analysis of storage systems. Tracing collects precise data about the internal behavior of the system, which helps in detecting performance problems that are difficult to identify.

The goal of this thesis is to provide a tool to analyze storage performance based on low-level trace events. The main challenges addressed by this tool are: collecting the required data using kernel and userspace tracing, limiting the overhead of tracing and the size of the generated traces, synchronizing the traces collected from different sources, providing multi-level analyses covering several aspects of storage performance, and lastly proposing abstractions allowing users to easily understand the traces.

We carefully designed and inserted the instrumentation needed for the analyses. The tracepoints provide full visibility into the system and track the lifecycle of storage requests, from creation to processing. The Linux Trace Toolkit Next Generation (LTTng), a free and low-overhead tracer, is used for data collection. This tracer is characterized by its stability, and efficiency with highly parallel applications, thanks to the lock-free synchronization mechanisms used to update the content of the trace buffers. We also contributed to the creation of a patch that allows LTTng to capture the call stacks of userspace events.

Achieving a balance between the accuracy of the analysis and the tracing overhead is a major challenge in our research. For each proposed analysis, we perform a wide range of tests to evaluate the overhead of the tracer, by computing its impact on the execution time

of the observed program. The size of the generated traces is another aspect of the tracing overhead that is important to consider. In this regard, we propose two mechanisms to avoid generating huge traces that are difficult to analyze. The first mechanism is based on two tracing sessions that operate simultaneously: lightweight tracing and exhaustive tracing. The lightweight tracing session traces a small number of events and analyzes them on the fly in order to to detect unusual behaviors. At the same time, the exhaustive tracing session writes the complete trace temporarily in a circular buffer. When an anomaly is detected by the lightweight tracing session, the content of the circular buffer is saved on disk and used for an in-depth analysis. The second mechanism proposed uses a dynamic sampling algorithm to reduce the frequency of event generation. Instead of recording all the events, our algorithm aggregates them in kernel space and generates an event only if a certain condition is met.

Our tool is able to analyze traces collected from multiple machines and data sources. The traces are synchronized based on the monotonic clock of the kernel, and the causality between the events in the case of a distributed system. Trace analysis is done in two main phases. When the trace is read for the first time, a history database is created to keep the states of system components over time. This database is then used to provide the necessary information for the analysis algorithms, which considerably improves the interactivity of our tool.

This methodology was used to provide several types of analysis. First, we study the performance of mass storage devices based on the trace events generated by the Linux storage subsystem. Next, we extend this analysis to support distributed storage systems, principally Ceph, using the userspace instrumentation of storage daemons. Then, we provide an analysis that monitors the memory usage of system processes. Finally, we study dynamic memory allocation mechanisms and we analyze the performance of the Java garbage collector.

The use cases and the experimental results presented in this thesis prove the effectiveness of the approach. The proposed algorithms are implemented as plugins for Trace Compass, a free trace analysis tool. We provide performance metrics and graphical views to facilitate the analysis of the observed system.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

API	Application Programming Interface
BIO	Block Input/Output
CERN	Conseil Européen pour la Recherche Nucléaire
CFQ	Complete Fair Queueing
CPU	Central Processing Unit
CRUSH	Controlled Replication Under Scalable Hashing
CTF	Common Trace Format
CephFS	Ceph File System
E/S	Entrée/Sortie
ETW	Event Tracing for Windows
EXT	Extended File System
FCFS	First Come First Served
FSM	Finite State Machine
GTF	Generalized Trace Facility
HDD	Hard Disk Drive
I/O	Input/Output
IOPS	Input/Output Operations per second
JIT	Just-In-Time
JMX	Java Management Extension
JVM	Java Virtual Machine
JVMTI	Java Virtual Machine Tool Interface
LRU	Least Recently Used
LTTng	Linux Trace Toolkit Next Generation
MDS	Metadata Server
MIB	Management Information Base
MMU	Memory Management Unit
MSS	Modeled State System
MTV	Memory Trace Visualizer
NFS	Network File System
NUMA	Non-uniform Memory Access
OSD	Object Storage Daemon
OTS	Off-The-Shelf
RAM	Random Access Memory

RBD	Rados Block Device
RCU	Read Copy Update
RamFS	RAM Filesystem
SHT	State History Tree
SMP	Symmetric Multiprocessing
SNMP	Simple Network Management Protocol
SSD	Solid-State Drive
TID	Thread Identifier
UST	Userspace Tracing
VFS	Virtual File System
WAF	Write Amplification Factor
WPA	Windows Performance Analyzer
WPR	Windows Performance Recorder

CHAPITRE 1 INTRODUCTION

Assurer la rapidité et le bon fonctionnement des systèmes informatiques est une exigence vitale dans l'industrie. Dans quelques domaines essentiels, comme les secteurs bancaire et financier, un problème de performance peut coûter des milliards de dollars ou même causer la faillite d'une entreprise.

Un grand effort a été fourni au niveau de la microarchitecture pour concevoir des processeurs rapides et performants. Les processeurs modernes contiennent plusieurs cœurs physiques, et chacun des cœurs est muni de plusieurs unités de calcul intégralement pipelinées. Ces optimisations ont permis de fournir une énorme puissance de calcul, mais leur impact sur la performance globale du système n'est pas toujours garanti. En effet, si une application passe la majorité du temps bloquée sur des opérations de stockage, l'utilisation d'un processeur plus rapide n'a presque aucun effet sur son temps d'exécution.

Plusieurs mécanismes ont été proposés dans le but d'améliorer la performance de stockage. L'analyse de l'efficacité de ces solutions nécessite l'étude approfondie du comportement de l'application et du système d'exploitation au moment de l'exécution.

Notre objectif général est de développer des méthodes permettant d'analyser les performances de stockage, en mémoire et sur les périphériques d'E/S, et d'étudier leur impact sur la rapidité des systèmes en production.

1.1 Définitions et concepts de base

Les opérations de stockage représentent un goulot d'étranglement dans les systèmes informatiques à haute performance. La latence occasionnée par ces opérations est généralement grande et cause le blocage des programmes. L'utilisation de plusieurs fils d'exécution peut aider à cacher le temps de blocage, mais ceci n'est pas toujours possible, car les opérations de stockage peuvent se trouver sur le chemin critique de l'application [1].

Plusieurs mécanismes de gestion ont été implémentés au sein des systèmes d'exploitation pour améliorer les performances de stockage. Les ordonnanceurs de disque permettent de réorganiser les requêtes d'E/S pour assurer une meilleure utilisation des périphériques de stockage de masse. Le choix de l'algorithme d'ordonnancement dépend du critère de performance choisi. On peut par exemple augmenter le débit du disque en diminuant la fréquence des accès aléatoires, organiser les requêtes selon la priorité des processus, etc. Le cache du système de fichiers est un autre mécanisme qui permet d'éviter les accès répétés au disque

en gardant les données les plus utiles dans la mémoire principale.

Le stockage efficace des données en mémoire est un autre facteur important à considérer. La séparation conceptuelle entre l'adressage physique et virtuel a permis d'optimiser l'utilisation de la mémoire et de faciliter le partage des données entre les processus. Un composant logiciel, appelé allocateur de mémoire, est utilisé pour optimiser le placement des objets dans la mémoire. Certains langages de programmation populaires comme Java et C# ont totalement éliminé la gestion manuelle de la mémoire grâce au ramasse-miettes, un composant logiciel permettant de libérer les objets inutiles sans l'intervention explicite du programmeur.

Les mécanismes de gestion de la mémoire et des périphériques d'E/S ont permis de notablement améliorer les performances de stockage. Cependant, le niveau de complexité supplémentaire introduit par ces mécanismes a rendu le débogage encore plus difficile. Par exemple, le temps de traitement d'une requête de disque est affecté par plusieurs composants : la mémoire cache, le système de fichiers, l'ordonnanceur du disque, le pilote, etc. En cas d'anomalie, il serait difficile de savoir la cause qui l'a engendrée.

Un autre aspect à scruter est la dépendance entre la performance des opérations de stockage et la charge de travail réelle appliquée. Par exemple, l'efficacité de l'ordonnanceur de disque dépend fortement des caractéristiques et de la répartition des requêtes d'E/S à traiter. De même, l'efficacité d'un allocateur de mémoire dépend de la taille et de la répartition temporelle et spatiale des objets alloués. Partant de ce fait, une étude approfondie des caractéristiques du système cible est nécessaire pour pouvoir obtenir les meilleures performances.

Plusieurs outils peuvent être utilisés pour analyser les performances des systèmes informatiques, mais ces outils présentent des limitations dans le contexte de l'étude de la performance de stockage. **Les outils de monitorage** permettent de suivre l'utilisation des ressources matérielles (mémoire, réseau, périphériques de stockage, etc.) en effectuant de la scrutation périodique des statistiques générées par le système d'exploitation. La période de scrutation est généralement fixée à une seconde afin d'éviter le surcoût de la collecte des statistiques. Cette période permet de donner une idée globale sur l'état du système, mais ne permet pas de détecter les anomalies, car les cas spéciaux sont absorbés dans la moyenne.

Les outils d'échantillonnage génèrent des informations sur la distribution des fonctions d'un programme donné. La distribution peut être temporelle ou selon un autre critère comme les défauts de cache ou le nombre d'instructions. Cette information permet de surligner les fonctions d'intérêt et de guider le développeur dans le processus de débogage. Les outils d'échantillonnage prennent en compte les instructions exécutées sur le processeur et ignorent les périodes de blocage, ce qui les rend inefficaces dans l'analyse des latences d'accès à la mémoire et aux périphériques d'E/S.

Un débogueur permet d'examiner des problèmes au niveau de la logique des programmes. Il est capable d'exécuter le programme instruction par instruction et d'accéder au contenu des variables au moment de l'exécution. Les débogueurs sont très efficaces pour résoudre certains types de problèmes comme les accès illégaux à la mémoire, les divisions par zéro, les boucles infinies, etc. Par contre, ils sont incapables de détecter des anomalies peu fréquentes et difficiles à reproduire. Par exemple, une condition de concurrence critique qui implique plusieurs fils d'exécutions est très difficile à dépister avec un débogueur. L'utilisation d'un débogueur n'est pas appropriée dans le cadre de l'étude des performances de stockage, car les problèmes sont généralement imprévisibles et causés par des facteurs externes au programme.

Les outils d'étalonnage sont très populaires dans l'analyse des performances de stockage. Des charges de travail de caractéristiques différentes sont exécutées sur le système afin d'étudier son fonctionnement avec un large éventail de configurations. Les outils d'étalonnage présentent deux grandes limitations. Premièrement, les charges de travail utilisées sont synthétiques et ne reflètent pas le fonctionnement du système dans un contexte réel [2]. Deuxièmement, ces outils n'offrent pas une visibilité sur les différents composants du système, et ainsi ils sont incapables de trouver les raisons principales d'une dégradation de performance.

Le traçage est un mécanisme qui consiste à enregistrer les événements d'intérêt qui se produisent lors de l'exécution du système. Un événement est un point marquant et ponctuel dans l'exécution d'un programme caractérisé par son type et son estampille de temps. Il peut par exemple indiquer un appel système, un appel de fonction ou une interruption. Chaque événement vient avec une charge utile qui contient des informations sur l'état du système au moment de son occurrence.

Le traçage des applications nécessite une étape préliminaire d'instrumentation pendant laquelle on précise les points de trace nécessaires à l'analyse. Un point de trace définit l'emplacement précis, dans le code source de l'application, auquel un événement doit se produire. L'instrumentation peut se faire statiquement, en insérant les points de traces au moment de la compilation, ou dynamiquement pendant l'exécution. Le traçage peut se faire dans l'espace utilisateur ou dans l'espace noyau. La trace de l'espace utilisateur permet de générer des événements proches de la logique du programme. Le traçage en mode noyau génère des événements de très bas niveau, fournissant des informations sur le fonctionnement interne du système d'exploitation.

Contrairement aux techniques d'étalonnage, le traçage permet d'étudier le système avec des charges de travail réelles, ce qui permet d'étudier la performance des systèmes en production. D'autre part, le traçage permet d'analyser le comportement de plusieurs composants simultanément. Par exemple, il est possible de collecter des données depuis le système de

fichiers, l'ordonnanceur du disque, le pilote du disque, les interruptions système et d'utiliser la trace générée pour avoir une évaluation globale sur l'efficacité du sous-système de stockage. Un autre avantage de traçage est sa capacité de détecter des inefficacités difficilement identifiables. Puisqu'on enregistre l'historique détaillé de tous les événements produits dans le système, il est possible de capturer des anomalies rares et difficiles à reproduire.

Dans cette thèse, nous utilisons LTTng [3], un traceur à faible surcoût disponible sur Linux, pour étudier les performances de stockage. Notre recherche s'intéresse à la collecte et à l'exploitation des données générées par le traceur.

1.2 Éléments de la problématique

Le traçage fournit des informations très utiles sur les systèmes en exécution, mais par contre son utilisation présente plusieurs défis technologiques.

Le traçage peut engendrer le ralentissement de l'exécution si le mécanisme d'instrumentation n'est pas optimal ou si les points de trace sont insérés dans une source d'événements à débit élevé. Le surcoût du traçage est parmi les grands enjeux de notre recherche. Il est nécessaire de mesurer le surcoût du traçage et de proposer des techniques pour le limiter. La réduction du nombre de points de trace actifs permet de diminuer le surcoût du traçage, mais elle affecte la précision de l'analyse. Atteindre un équilibre entre la précision de l'analyse et le surcoût du traçage est un des compromis à considérer dans notre recherche.

L'étude des performances de stockage nécessite une instrumentation particulière du système d'exploitation. L'instrumentation existante dans le noyau Linux couvre la plupart des aspects nécessaires pour l'analyse. Si une instrumentation additionnelle est requise, il faut proposer des méthodes pour éviter la tâche délicate de recompiler le noyau à partir des sources. Le même défi se pose pour l'instrumentation des applications. La collecte des informations de l'espace utilisateur nécessite l'instrumentation des applications et donc la modification de leur code source. L'instrumentation pourrait se faire dynamiquement, en chargeant une librairie au moment du lancement du programme, mais cette solution n'est pas possible si l'application observée utilise des mécanismes de protection de l'espace d'adressage.

Un autre défi se présente lors de l'analyse des performances des systèmes de stockage distribués. Les machines possèdent des horloges différentes et l'analyse globale du système nécessite la synchronisation des traces sur une base de temps commune. La précision des algorithmes de synchronisation dépend du nombre d'événements liés sémantiquement, et ceci peut causer une incertitude dans nos analyses. L'utilisation des événements d'envoi et de réception de paquets réseau permet d'avoir un niveau de précision acceptable si le nombre de paquets

échangés est suffisamment grand.

Les événements inscrits dans la trace sont généralement très détaillés et difficiles à comprendre. Quelques événements de trace ne sont pas significatifs individuellement et nécessitent la connaissance du contexte pour être interprétés. Un des buts de notre recherche est de fournir des abstractions permettant de faciliter la compréhension des traces. Les abstractions peuvent être des abstractions visuelles, soit des vues graphiques qui modélisent le système observé, ou sous forme de statistiques et de métriques de performance. Des structures de données adéquates doivent être choisies pour garder l'historique de l'état du système au cours de la session du traçage. Ces structures de données doivent être à la fois rapides et extensibles, vu la grande taille des traces à traiter.

1.3 Objectifs de recherche

L'objectif général de cette recherche est de fournir des outils permettant d'analyser les performances de stockage, en mémoire et sur les périphériques d'E/S, à partir d'une trace d'exécution.

Nous avons défini les objectifs particuliers suivants :

- Produire l'instrumentation nécessaire pour les analyses.
- Mesurer le surcoût du traçage et proposer des méthodes pour réduire la taille des traces générées.
- Fournir des analyses qui couvrent plusieurs aspects de la performance de stockage.
- Fournir des abstractions permettant de mieux représenter les données.

1.4 Contributions

Pour atteindre les objectifs de recherche ci-dessus, cette thèse présente les contributions originales suivantes dans le domaine d'étude des performances de stockage.

- Une méthode de traçage efficace pour les systèmes distribués. L'idée est de se servir d'une session de traçage légère, qui utilise un nombre réduit de points de trace, pour avoir une vue globale sur la santé des machines de la grappe. Lorsqu'un problème de performance est détecté, les traces détaillées sont capturées et utilisées pour trouver la cause racine du problème (Section 5.4.1).
- Un mécanisme d'échantillonnage dynamique permettant de tracer efficacement une source d'événements à débit élevé. L'algorithme proposé fait des agrégations du côté noyau et génère un événement de trace seulement si une certaine condition est remplie (Section 6.5.2).

— Des algorithmes efficaces permettant d'étudier les performances de stockage en se basant sur des traces d'exécution (Sections 4.5, 5.5 et 7.5). Les analyses proposées supportent les événements émis par le noyau et les programmes en mode utilisateur, ce qui permet d'avoir une visibilité complète sur le système observé. Des interfaces graphiques interactives sont générées pour faciliter la compréhension et l'analyse des traces.

1.5 Plan

La revue de la littérature est présentée au Chapitre 2. On y présente les travaux antérieurs relatifs à notre sujet de recherche. La méthodologie utilisée pour répondre aux objectifs de recherche est présentée dans le Chapitre 3. Les quatre articles scientifiques issus de la recherche suivent successivement.

L'article "Recovering Disk Storage Metrics from Low-level Trace Events" au Chapitre 4 porte sur l'analyse des performances des périphériques de stockage de masse. Cet article a été publié dans le journal *Software : Practice and Experience*.

L'article "Performance Analysis of Distributed Storage Clusters Based on Kernel and Userspace Traces" au Chapitre 5 porte sur l'analyse des performances des systèmes de stockage distribués. Cet article a été soumis au journal *Software : Practice and Experience*.

L'article "Dynamic Trace-based Sampling Algorithm for Memory Usage Tracking of Enterprise Applications" au Chapitre 6 propose un mécanisme d'agrégation du côté noyau permettant de réduire le surcoût du traçage des événements de mémoire. Cet article a été présenté à la conférence *IEEE High Performance Extreme Computing Conference*.

L'article "Multilevel Analysis of The Java Virtual Machine Based on Kernel and Userspace Traces" au Chapitre 7 porte principalement sur la performance des ramasse-miettes utilisés par la machine virtuelle Java. Cet article a été soumis au journal *Journal of Systems and Software*.

La thèse se termine avec la discussion générale au Chapitre 8 et la conclusion au Chapitre 9.

CHAPITRE 2 REVUE DE LITTÉRATURE

Ce chapitre présente les concepts nécessaires à la compréhension de la thèse et décrit l'état de l'art dans ce domaine. Le but est de positionner notre recherche dans le cadre des travaux existants afin de démontrer sa pertinence et son originalité.

2.1 Stockage de masse

L'objectif principal des périphériques de stockage de masse est le stockage d'une grande quantité d'informations à long terme. Le sous-système de stockage du système d'exploitation Linux contient des composants permettant d'assurer la gestion optimisée de ces périphériques.

Dans cette section, nous commençons par décrire l'architecture et l'implémentation du sous-système de stockage. Ensuite, nous présentons les méthodes d'analyse de performance des périphériques de stockage de masse.

2.1.1 Architecture du sous-système de stockage sous Linux

Les opérations d'accès aux périphériques de stockage de masse ont souvent été considérées comme le goulot d'étranglement dans les systèmes haute performance. Les concepteurs des systèmes d'exploitation ont investi un grand effort dans l'implémentation d'un sous-système de stockage modulaire permettant d'avoir de bonnes performances avec différents types de périphériques de stockage tels que les disques magnétiques, les disques SSD et les mémoires flash. Les composants du système d'exploitation impliqués dans ce sous-système sont illustrés à la figure 2.1.

Une opération d'accès au disque passe par plusieurs étapes pour être servie. Elle commence au niveau de l'espace utilisateur, lorsque le programme exécute un appel système de lecture ou d'écriture. Ces appels sont directement liés au VFS (Virtual File System) [4], une interface générique qui cache les particularités des systèmes de fichiers. Si l'opération ne peut pas être directement servie à partir du cache, elle est convertie par le système de fichiers en plusieurs requêtes pour qu'elles soient traitées par la couche des périphériques de stockage. L'ordonnanceur de disque garde les requêtes dans des files d'attente et utilise des algorithmes d'ordonnancement pour sélectionner les requêtes à traiter. Enfin sélectionnée, la requête est envoyée au disque par l'intermédiaire du pilote, un module noyau qui implémente le protocole d'accès au périphérique.

Les prochains paragraphes discutent plus en détail le rôle et l'implémentation des composants

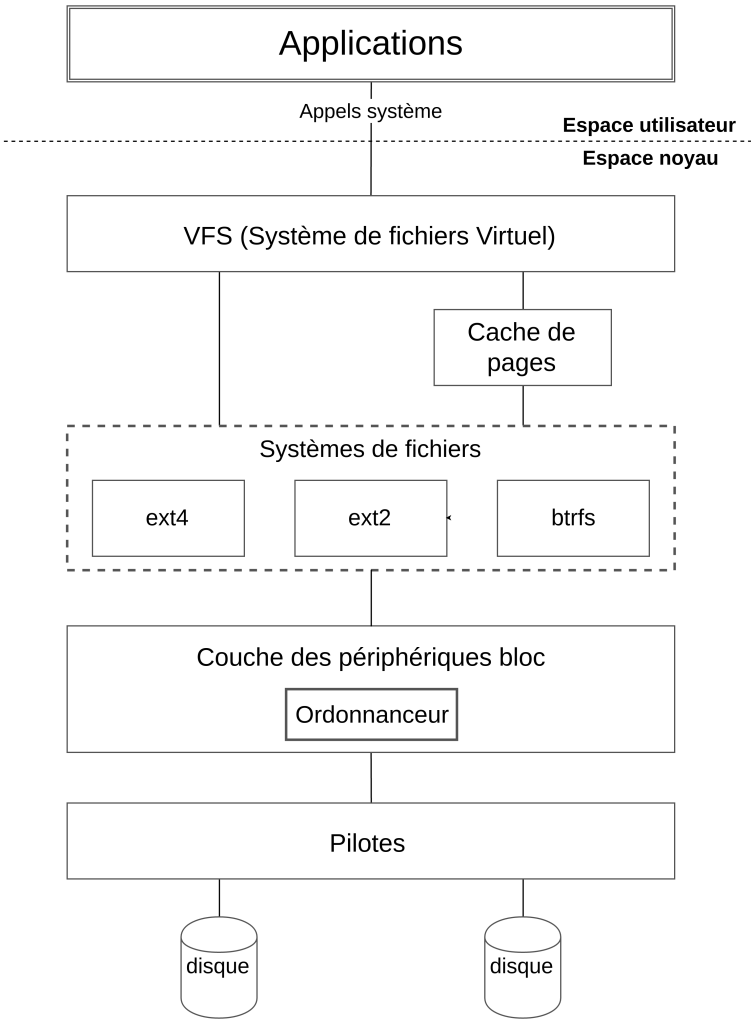


Figure 2.1 Architecture du sous-système de stockage

du sous-système de stockage.

Périphériques bloc

Les périphériques bloc sont utilisés principalement pour le stockage de données [5]. Contrairement aux périphériques par caractères (flots d'octets), ils offrent la possibilité d'accéder à l'information d'une manière aléatoire. Les opérations de lecture et d'écriture peuvent toucher n'importe quelle adresse, sans contrainte sur l'ordre d'accès. Pour faciliter la localisation des données, l'espace de stockage est divisé en plusieurs segments de petite taille, appelés des *secteurs*. La taille d'un secteur est généralement entre 512 octets et 4 Kio, c'est la plus petite unité adressable par le périphérique. Le système de fichiers, par contre, utilise une autre unité d'adressage logique qui s'appelle *bloc*. La taille d'un bloc doit obligatoirement être un

multiple de la taille d'un secteur. Puisque les blocs sont enregistrés dans des pages mémoires, la taille d'un bloc doit aussi être plus petite que la taille d'une page.

BIO (Block Input/Output) est la structure de données principale utilisée pour effectuer des opérations d'E/S sur un périphérique bloc [5]. Un BIO est formé de plusieurs segments élémentaires, chacun représenté par *bio_vec*. Chaque segment est défini par un triplet (page, déplacement, taille) qui indique la page mémoire où le tampon existe, le déplacement dans la page, et la taille du tampon (Figure 2.2).

Cette implémentation modulaire donne à BIO la flexibilité de faire du *scatter-gather*. En d'autres termes, BIO est capable de définir un tampon d'E/S éparpillé sur plusieurs pages mémoire, ce qui était impossible avec la structure *buffer_head* utilisée dans les versions de Linux antérieures à la révision 2.5.

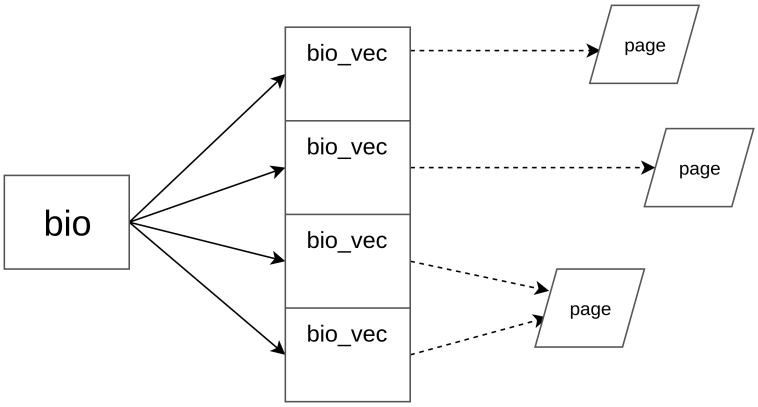


Figure 2.2 La structure de données BIO (Block Input/Output)

Le contrôleur du disque manipule les opérations d'E/S sous forme de requêtes. Une requête est composée d'un ensemble de *BIO* ciblant des secteurs physiquement adjacents. Une requête d'E/S peut être de lecture ou d'écriture, synchrone ou asynchrone, etc. L'ordonnanceur du disque peut fusionner deux requêtes adjacentes, seulement si elles sont de même type. Le périphérique de stockage maintient une file contenant les requêtes en attente de traitement. Cette file est implémentée en utilisant une liste doublement chaînée pour réduire la complexité des algorithmes d'ordonnancement. Un schéma donnant la structure générale d'une file d'attente est fourni dans la figure 2.3.

Algorithmes d'ordonnancement

Le disque magnétique, qui est le standard des périphériques de stockage, souffre d'une limitation majeure : le déplacement du bras mécanique est une opération très lente. Par conséquent,

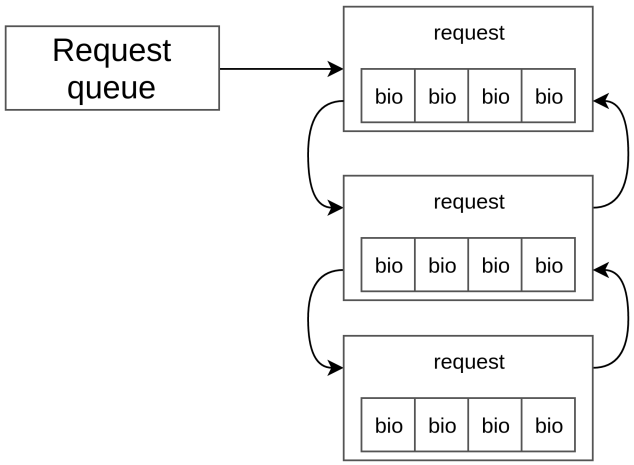


Figure 2.3 L'implémentation de la file d'attente du disque

le temps de positionnement est beaucoup plus grand que le temps de transfert de données. Historiquement, les ordonnanceurs de disque ont été créés pour diminuer le déplacement de la tête du disque, en triant et en fusionnant les requêtes.

Les premiers algorithmes d'ordonnancement étaient relativement simples. L'idée était tout simplement de trier les requêtes selon leurs secteurs dans un ordre croissant ou décroissant pour diminuer au maximum le va-et-vient de la tête de lecture et améliorer le débit. Par contre, le temps de réponse de ces algorithmes est très variable, puisqu'il dépend du secteur de la requête insérée. Si par hasard le secteur de la requête est proche de la position actuelle de la tête de lecture, le temps de réponse est très court. Sinon, la requête devra attendre longtemps avant d'être servie.

Les algorithmes d'ordonnancement modernes sont beaucoup plus sophistiqués. Les critères de performance utilisés pour l'évaluation de ces algorithmes sont variés et dépendent des caractéristiques du système. Les critères les plus utilisés sont le débit du disque, l'équité entre les processus, le respect des contraintes temps réel, etc. Plusieurs ordonnanceurs viennent par défaut avec le système d'exploitation Linux : *Noop*, *Deadline*, *Anticipatory*, et *CFQ*.

Noop [6] : Un ordonnanceur de type FCFS (First Come First Served). Les requêtes sont traitées dans le même ordre d'insertion. Les requêtes adjacentes de même type sont fusionnées pour accélérer leur traitement par le disque. Cet ordonnanceur est conçu pour être utilisé avec les disques basés sur les circuits intégrés comme les SSDs et les mémoires flash, car ils sont performants avec les charges de travail aléatoires.

Deadline [7] : Cet ordonnanceur assure que les requêtes sont traitées avant leur date d'échéance. L'implémentation de cet ordonnanceur se base sur quatre files d'attente. Deux files sont utilisées pour garder les requêtes d'E/S triées selon les secteurs et

les deux autres files gardent les mêmes requêtes triées selon les délais. Les requêtes de lectures et d'écritures sont gardées séparément. L'ordonnanceur donne toujours la priorité aux requêtes de lecture. La récupération des requêtes se fait à partir des files triées par secteur, sauf si l'ordonnanceur détecte une requête sur le point de dépasser son échéance. Dans ce cas, il récupère cette requête à partir de la file triée par échéance et l'envoie au disque.

Anticipatory [8] : Cet ordonnanceur est similaire à Deadline. Il utilise la même architecture, mais avec un algorithme d'ordonnancement légèrement différent. Dans l'ordonnanceur Deadline, une requête de lecture peut interrompre une charge de travail séquentielle d'écriture. Dans ce cas, la tête de lecture doit faire un va-et-vient pour servir cette requête et reprendre l'activité d'écriture. Si cette situation se répète fréquemment, le débit du disque sera réduit considérablement. Pour éviter une telle situation, une requête de lecture doit attendre un certain délai dans la file avant d'être délivrée au disque. Ceci augmente la probabilité d'avoir des requêtes adjacentes à fusionner et permet donc de traiter plusieurs opérations de lecture avant de reprendre le travail d'écriture. [8] Cet algorithme a un bon débit global, mais il donne un mauvais temps de réponse pour les requêtes de lecture individuelles.

CFQ (Complete Fair Queueing) [9] : Contrairement aux autres algorithmes d'ordonnancement, cet ordonnanceur garde les requêtes dans des files par processus et il sélectionne les requêtes selon leur priorité. Les processus de haute priorité sont servis rapidement, et les processus de même priorité sont servis en *round-robin*. Les requêtes asynchrones sont gardées ensemble dans d'autres files, une file par priorité. Cet ordonnanceur est parfait pour les systèmes dont la charge de travail des processus est connue à l'avance. Il donne aussi des performances acceptables avec la plupart des charges de travail. C'est pour cette raison qu'il est choisi comme l'ordonnanceur de défaut de Linux.

VFS et Systèmes de fichiers

Le système d'exploitation Linux permet d'interagir avec plusieurs types de systèmes de fichiers d'une manière transparente. Cette flexibilité est offerte par l'intermédiaire du système de fichiers virtuel (VFS) qui fournit une interface commune à tout les systèmes de fichiers. Une application peut donc exécuter des opérations d'E/S sans se soucier du type du système de fichiers ou de l'architecture du périphérique de stockage.

VFS introduit un modèle de fichiers permettant de définir une organisation logique des données totalement indépendante de leurs dispositions physiques. Ce modèle est basé sur quatre

structures de données principales :

superblock : représente un système de fichiers.

inode : représente un fichier. Il contient des informations sur la disposition physique du fichier sur le disque. Chaque fichier est identifié par un *inode* unique.

file : représente un fichier ouvert par un processus. Cette structure de données permet de modéliser la relation N :N entre les fichiers et les processus.

dentry : garde la correspondance entre le nom de fichier et l'*inode* associé.

Ce modèle générique a permis à Linux de supporter la plupart des systèmes de fichiers existants, même les plus anciens. On peut distinguer trois catégories principales de systèmes de fichiers.

— *Systèmes de fichiers locaux :* Ce sont les systèmes de fichiers ordinaires qui gèrent l'accès aux fichiers stocké sur un disque de stockage, par exemple : Ext2 [5], Ext4 [10], Btrfs [11], etc.

— *Systèmes de fichiers distribués :* Ils permettent d'accéder à des fichiers stockés sur des ordinateurs distants. Les opérations d'E/S sont envoyées par réseau vers le système de fichiers distant qui traite les requêtes et retourne le résultat. Ceci se fait d'une manière totalement transparente à l'utilisateur, par exemple : NFS [12], AFS [13], etc.
Les systèmes de stockage distribué sont présentés en plus de détails dans la Section 2.2.

— *Systèmes de fichiers spéciaux :* Les systèmes de fichiers spéciaux permettant d'interagir avec le noyau du système d'exploitation à partir de l'espace utilisateur, par exemple : Procfs, Sysfs, etc.

Dans le reste de cette section, nous décrivons brièvement Ext2 et Ext4, deux systèmes de fichiers locaux natifs au système d'exploitation Linux.

Ext2 : Ext2 [5] est un système de fichiers développé en 1993 pour remplacer Ext (Extended File System). Il a été longtemps considéré comme le système de fichiers standard de Linux vu le grand nombre d'améliorations qu'il propose. L'espace de stockage maximal supporté par Ext2 est 4Tio, et la taille maximale d'un fichier est 2Gio. Il offre la possibilité de choisir une taille de *bloc* entre 1Kio et 4Kio. L'*inode* du fichier contient des pointeurs directs vers les blocs physiques, ou des pointeurs indirects hiérarchiques si le fichier est de grande taille (figure 2.4). Ext2 garde plusieurs copies de la table des *inodes* pour faciliter la récupération des fichiers en cas de plantage du système.

Ext4 : Ext4 [10] est le système de fichiers par défaut dans les versions récentes de Linux. L'architecture de Ext4 est similaire à Ext2, ce qui assure la compatibilité entre les deux systèmes de fichiers. Ext4 supporte un espace de stockage de 1Eio et une taille de fichier de 16Tio, ce qui est largement suffisant même dans les grands centres de données. La fonctionnalité la plus intéressante de Ext4 est la journalisation, qui consiste à garder l'historique des opérations d'E/S dans un journal, ce qui permet de remettre le système de fichiers dans un état stable après un plantage brusque du système. Trois modes de journalisation sont disponibles. En mode *journal*, les données et les métadonnées des fichiers sont écrites dans le journal avant de modifier l'état du fichier sur le disque. En mode *ordered*, seulement les métadonnées sont écrites dans le journal. Les métadonnées ne sont écrites dans le journal que lorsque l'opération d'E/S est achevée. Le mode *write-back* est similaire à *ordered*, sauf qu'il n'y a pas un ordre d'écriture imposé.

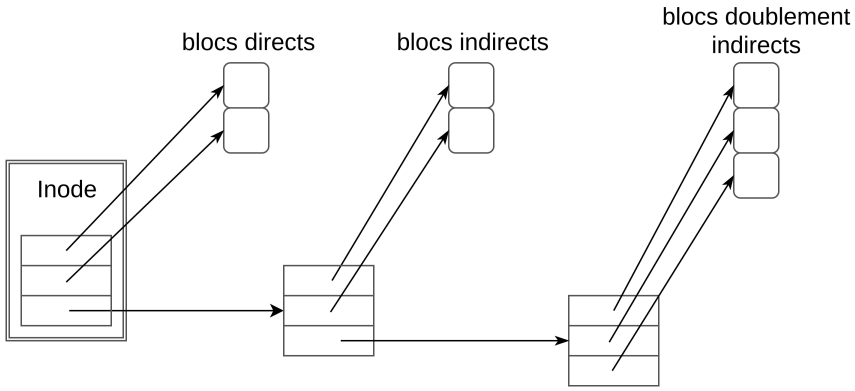


Figure 2.4 La structure Inode utilisée dans les systèmes de fichiers Ext2 et Ext4

2.1.2 Méthodologies d'analyse des performances de stockage de masse

Dans cette section, nous présentons un résumé des méthodologies d'analyse les plus utilisées dans la littérature. Le but est de donner une vue globale du domaine, ce qui permet de mieux classifier les travaux de recherche par la suite. Nous avons distingué trois méthodologies principales d'étude de performance des périphériques de stockage.

La première méthodologie (Figure 2.5) consiste à utiliser des outils d'étalonnage pour évaluer la performance de stockage. L'étalonnage consiste à exécuter des charges de travail sur le système et de calculer le temps d'exécution de chacune d'elle avec de différentes configurations. Par exemple, on peut comparer les accès séquentiels et aléatoires au disque ou étudier l'impact des tailles des requêtes sur les performances globales. Cette méthodologie souffre de deux inconvénients majeurs. Premièrement, les charges de travail utilisées pour l'étude

des performances sont synthétiques et ne reflètent pas le vrai fonctionnement du système. Deuxièmement, le seul critère de performance utilisé, le temps d'exécution, est très générique et ne permet pas de repérer les causes racines des défaillances.

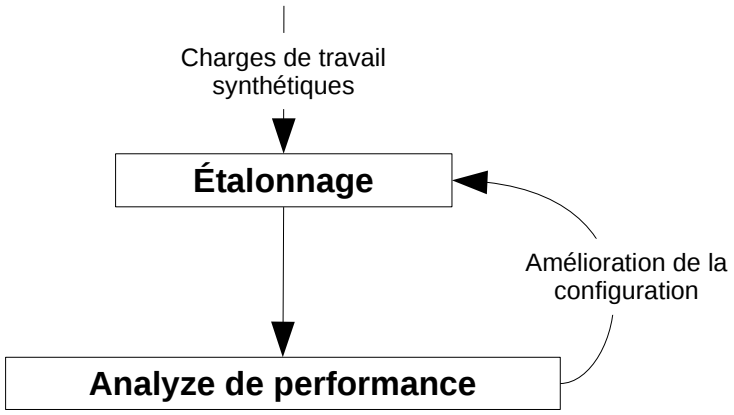


Figure 2.5 Méthodologie 1 : Étalonnage

La deuxième méthodologie (Figure 2.6) consiste à utiliser le traçage pour étudier les caractéristiques du système et comprendre les patrons d'accès. L'information collectée est utilisée pour générer des charges de travail similaires, ou identiques, à la charge de travail réelle.

Cette méthodologie améliore la précision de l'étalonnage, car elle prend en compte des caractéristiques du système étudié et utilise pas des charges de travail purement synthétiques. Par contre, comme la première méthodologie, elle ne permet pas de repérer les causes racines des problèmes de performance.

La troisième méthodologie (Figure 2.7) consiste à utiliser des outils de traçage pour collecter des informations précises sur le fonctionnement du système dans des conditions réelles. L'avantage de cette méthodologie est de pouvoir identifier les anomalies qui peuvent se produire dans des environnements de production. Les travaux de recherche pertinents doivent relever deux défis. Premièrement, il faut bien choisir le mécanisme de traçage afin d'éviter d'altérer le fonctionnement du système. Deuxièmement, les données générées par le traceur sont de grande taille et il faut proposer des outils automatiques pour pouvoir les analyser.

Les prochaines sections présentent les outils et les articles de recherche liés à chaque méthodologie et discutent les avantages et les limitations de ces travaux.

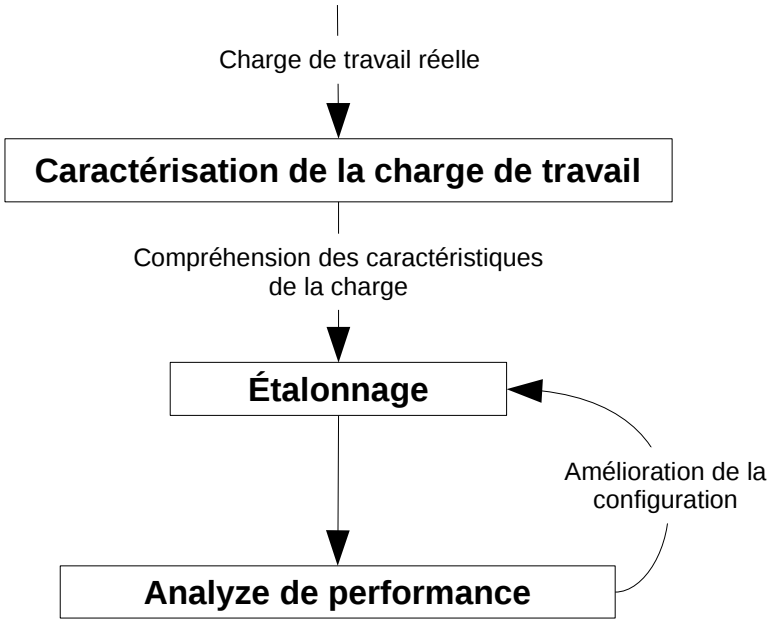


Figure 2.6 Méthodologie 2 : Caractérisation de la charge de la charge de travail dans le but d'améliorer la précision de l'étalonnage

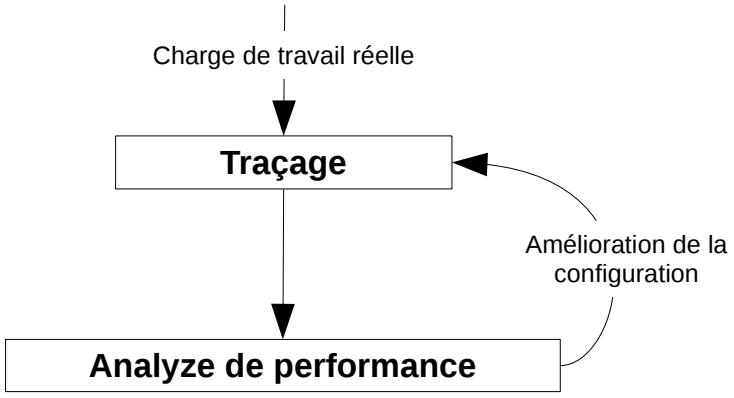


Figure 2.7 Méthodologie 3 : Traçage

2.1.3 Étalonnage

L'Étalonnage est un mécanisme très utilisé dans l'étude de performances de stockage. Il permet d'avoir une idée globale sur les performances du système avec différentes configurations. Vu la complexité du sous-système de stockage, l'étalonnage peut se faire sur plusieurs niveaux.

L'Étalonnage le plus basique consiste à manipuler directement des données sur le disque sans passer par le système de fichiers. En d'autres termes, les opérations de lecture ou d'écriture sont définies par les secteurs et ne sont pas affectées par le choix ou la configuration du système de fichiers. Ce type d'étalonnage est utile si le but est d'évaluer les performances du périphérique de stockage en tant que matériel, sans prendre en compte les mécanismes d'optimisation implémentés au sein du sous-système de stockage. IOmeter [14] est un des outils populaires permettant de réaliser ce type d'étalonnage.

Contrairement à IOmeter, IOzone [15] et Bonnie++ [16] effectuent les opérations de lecture et d'écriture sur des fichiers définis au niveau du système de fichiers. Ce type d'étalonnage évalue les performances du sous-système de stockage en totalité, incluant l'accès aux métadonnées et aux données [17].

Les utilisateurs des outils d'étalonnage veulent généralement définir les caractéristiques des charges de travail à exécuter. Fio [18] offre la possibilité de définir un fichier de configuration qui décrit les types d'opérations à exécuter, le nombre de fils d'exécutions qui roulent simultanément, la taille et la granularité des opérations, etc. Filebench [19] propose un langage de script que l'utilisateur peut prendre pour définir les propriétés de l'étalonnage. Ceci permet de créer différentes combinaisons intéressantes, qui mettent en relief plusieurs aspects de la performance du stockage.

L'étalonnage est très populaire dans l'étude de performance de stockage, grâce à sa simplicité. Néanmoins, les outils d'étalonnage peuvent donner des résultats trompeurs s'ils ne sont pas utilisés avec précaution [2]. Le sous-système de fichiers est très complexe et le temps d'exécution d'une charge de travail donnée peut être affecté par plusieurs facteurs. Il est difficile d'isoler l'effet de chaque composant sur le temps d'exécution total. Par exemple, un accès à un fichier nécessite d'abord un accès aux métadonnées stockées par le système de fichiers. Les métadonnées peuvent être dans la mémoire cache, ou pas, selon les opérations antérieures. De même, le mécanisme de prélecture peut affecter les résultats, alors que l'outil d'étalonnage n'a aucun contrôle sur un tel mécanisme. Traeger et al. [20] ont mené un sondage sur 415 étalonnages utilisés par 106 articles de recherche, et ils ont montré que plusieurs d'entre eux sont imprécis et ne donnent pas une idée claire sur la vraie performance du système. Tarasov et al. [2] ont fait une recherche similaire et ils ont montré que la mémoire cache cause un biais de mesure lors de l'étalonnage du système de fichiers.

2.1.4 Caractérisation de la charge de travail

Le développement d'un système de stockage est une tâche très compliquée. La performance d'un système de fichiers ou d'un algorithme d'ordonnancement est fortement liée à la charge

de travail appliquée. Il est impossible de concevoir un algorithme parfait pour toutes les situations ; il y a toujours des compromis à faire. Par exemple, les techniques de prélecture sont très efficaces avec les activités séquentielles d'E/S mais dégradent les performances des opérations d'E/S aléatoires. Une étude approfondie des caractéristiques de la charge de travail est une étape primordiale dans le choix des techniques de stockage les mieux adaptées au système.

Un des premiers travaux réalisés dans ce sens est celui de Ousterhout et al. [21] Les auteurs ont utilisé les traces du système de fichiers pour définir les patrons d'accès et générer des métriques aidant à comprendre les caractéristiques du système. Les fonctions tracées sont les suivantes : création de fichier, suppression de fichier, ouverture de fichier, fermeture de fichier et déplacement de curseur. L'analyse des traces a permis de mesurer le débit moyen, la quantité de données échangées, le nombre de fichiers manipulés, la durée de vie des fichiers, etc. La même étude a été reprise dans [22] [23] mais en tenant compte de plus d'événements, notamment la création des dossiers, le changement des permissions et l'utilisation des liens symboliques.

Les traces collectées peuvent être utilisées pour proposer des optimisations spécifiques au système étudié. Par exemple, Smith et al. [24] ont utilisé des techniques de simulation pour étudier l'impact du cache à partir de la trace. Les paramètres pris en compte dans l'étude sont la taille du cache, la taille du bloc, l'algorithme de remplacement, etc. Les traces sont collectées sur trois serveurs IBM avec le traceur GTF (Generalized Trace Facility). Contrairement aux études précédentes, la trace contient des événements bas-niveau collectés depuis la couche des périphériques bloc, pas seulement des appels système. Ruemmler et al. [25] ont fait la même étude sur des machines HP et ils ont montré que l'ajout d'une petite mémoire volatile à l'intérieur du contrôleur du disque permet d'améliorer considérablement les performances de stockage.

La résolution des chemins d'accès au fichier est une opération coûteuse. Chaque fois qu'un processus demande d'ouvrir un fichier, le système de fichiers doit parcourir les nœuds du chemin afin de trouver l'*inode* associé. Floyd et al. [26] ont montré en utilisant un traceur que le trafic de résolution du chemin d'accès constitue plus que 70% des accès au disque et que l'utilisation d'une mémoire cache permet d'éliminer ce trafic.

Les systèmes de fichiers distribués permettent de stocker des fichiers sur plusieurs machines d'une manière transparente à l'utilisateur. Lorsqu'une opération d'E/S touche un fichier local, l'information peut être facilement récupérée à partir du disque, tandis que si le fichier est placé sur un nœud distant, il faut passer par le réseau pour lui accéder. Dans les articles [27] et [28], les auteurs ont utilisé la caractérisation de la charge de travail pour résoudre le

problème de placement des fichiers dans les nœuds d'un système distribué. En se basant les événements de trace, il a été possible de proposer des algorithmes de migration des fichiers entre les nœuds qui minimisent les accès réseau.

Le placement des fichiers sur le disque est aussi un problème intéressant qu'on peut résoudre en utilisant le traçage. Stata [29] a développé un algorithme de migration de fichiers en se basant sur leur disposition physique. Les fichiers généralement accédés en séquence doivent être placés les uns après les autres sur le disque pour encourager les accès séquentiels. Staelin et al. [30] ont développé un système de fichiers intelligent qui utilise un algorithme similaire pour grouper les fichiers actifs durant l'exécution du système.

Strange [31] a travaillé sur le mécanisme de réplication des données dans un disque de sauvegarde. L'algorithme proposé se base sur les traces d'exécution pour récupérer les temps d'accès aux fichiers. Les fichiers sont triés selon leurs derniers temps d'accès. Les fichiers les plus récents restent sur le disque dur et les fichiers qui n'ont pas été accédés depuis longtemps sont migrés vers le disque de sauvegarde.

Gribble et al. [32] ont analysé des traces collectées sur quatre systèmes de fichiers et ils ont montré que les événements d'E/S sont auto similaires, ce qui veut dire que les requêtes d'E/S viennent généralement en rafale. Cette observation est très importante, car elle aide les administrateurs système à bien configurer la longueur des files d'attente du disque.

Agrawal et al. [33] ont proposé CodeRMI, un outil qui utilise la caractérisation de la charge de travail pour collecter des informations sur la répartition et les types des requêtes d'E/S traitées par le sous-système de stockage. Les données collectées sont par la suite utilisées pour créer des charges de travail synthétiques plus représentatives de la charge réelle, ce qui permet d'améliorer la précision des outils d'étalonnage. iGen [34] est un outil similaire qui génère des charges de travail pour des infrastructures de stockage distribué.

Au lieu de générer des charges de travail représentatives, d'autres outils utilisent la technique de relecture de trace pour imiter exactement la charge réelle. Ces outils enregistrent les estampilles de temps, les tailles et les types des requêtes d'E/S pour pouvoir reproduire le même comportement par la suite [35] [36]. Une étude menée par Pereira et al. [37] a montré que les mesures données par ces outils ne sont pas précises, car ils ne considèrent pas la dépendance entre les requêtes. En effet, le temps de traitement d'une opération d'E/S est fortement lié au temps de traitement des requêtes précédentes. Présumer que les estampilles de temps des requêtes restent les mêmes dans tous les systèmes est une hypothèse erronée.

Pour résoudre ce problème, Haghdoost et al. [38] ont proposé une heuristique permettant de spéculer les dépendances entre les requêtes. En se basant sur le graphe de dépendances

généré, il est possible prévoir le comportement de l'application originale avec des systèmes de caractéristiques différentes.

2.1.5 Traçage du sous-système de stockage

Avant la version 2.6 du noyau Linux, l'architecture du sous-système de stockage était très rigide. Les concepteurs du système d'exploitation ont décidé de redévelopper cette couche logicielle en utilisant des algorithmes et des structures de données avancés. La nouvelle implémentation est très flexible, mais plus difficile à déboguer. Blktrace est un traceur de requêtes développé par Jens Axboe, le mainteneur officiel du sous-système de stockage, pour analyser la performance de stockage. Blktrace écrit les traces sous format binaire pour assurer un faible surcoût. Les données sont générées en utilisant l'instrumentation statique existante sur Linux.

Seekwatcher [39] est un outil de visualisation des traces générées par blktrace. Il est le clone de l'outil TazTool, offert avec le système d'exploitation Solaris. Seekwatcher présente plusieurs métriques de performance. Le déplacement par seconde, le nombre d'opérations par seconde et le débit sont représentés sous forme de courbes. Le temps est affiché sur l'axe des abscisses et les métriques de performance sur l'axe des ordonnées. La position de la tête de disque au cours du temps est affichée dans une carte de densité. Les régions les plus denses sont celles où la tête de disque a passé la plupart du temps. IOprof [40] et BTT [41] offrent des fonctionnalités très similaires à Seekwatcher.

Oracle ZFS Storage Software [42] est une application disponible dans les serveurs d'Oracle pour analyser les performances du sous-système de stockage. L'application utilise le traceur dynamique DTrace pour collecter des données de bas niveau sur les périphériques de stockage, et offre une interface web pour visualiser les résultats en temps réel. Plusieurs vues intéressantes sont offertes à l'utilisateur. Le déplacement de la tête du disque est représenté sous forme de courbe, ce qui n'est pas très convivial vu la grande fréquence de déplacement de la tête de lecture. L'application présente la latence des requêtes dans une carte de densité, avec le temps sur l'axe des X et les latences sur l'axe des Y ; la teinte de couleur montre le nombre d'opérations d'E/S dans l'intervalle de temps correspondant. L'utilisation d'une carte de densité pour montrer la répartition des latences est une très bonne idée, comme l'explique Gregg [43]. Les méthodes classiques de visualisation, comme les courbes de moyennes ne montrent pas la distribution des latences, puisque les valeurs aberrantes sont perdues dans la moyenne. Dans une carte de densité, les latences maximales et minimales peuvent être facilement retrouvées. La latence moyenne peut être graphiquement retrouvée en identifiant les régions avec la teinte la plus sombre. Malgré le surcoût relativement élevé introduit

par DTrace, ZFS Storage Software peut être utilisé dans un environnement de production. Malheureusement, cet outil n'est pas disponible sur Linux, le système d'exploitation le plus utilisé dans l'industrie.

Windows Performance Analyzer (WPA) est un outil d'analyse de performance inclus dans *Windows Assessment and Deployment Kit* [44]. Avec un très grand nombre de vues graphiques, WPA couvre presque toutes les métriques nécessaires pour une étude de performance de stockage. Les vues sont générées à partir des traces ETW (Event Tracing for Windows) produites par *Windows Performance Recorder* (WPR). L'utilisateur peut récupérer la liste détaillée de toutes les requêtes d'E/S traitées dans un intervalle de temps précis. Cette liste peut être triée selon plusieurs critères comme la latence, le processus parent et le temps d'insertion. Les métriques de performance offertes sont le déplacement par seconde, le nombre d'opérations par seconde, le débit, l'utilisation du disque, etc. WPA fonctionne seulement sur Windows et n'est pas compatible avec les autres systèmes d'exploitation.

Rodeh et al. [45] ont proposé un autre outil de visualisation pour les serveurs de données *XIV* de IBM. *XIV* vient avec un système de traçage propriétaire permettant de collecter des données sur le sous-système de stockage et de les sauvegarder dans tampon circulaire. Un processus externe récupère les traces périodiquement à partir du tampon, avant qu'elles soient écrasées. La visualisation d'une trace de longue durée est le défi principal relevé dans cet article. Les auteurs proposent de décomposer la trace en plusieurs cellules temps/espace. Chaque cellule est représentée dans un plan 2D par un cercle dont le rayon est proportionnel à la métrique calculée.

2.2 Systèmes de stockage distribué

Les exigences sur le stockage de masse ont beaucoup évolué dans la dernière décennie. Les entreprises ont besoin de systèmes de stockage extensibles, performants et faciles à maintenir, ce qui n'est pas le cas pour les architectures centralisées.

L'utilisation d'une architecture distribuée permet de fournir un modèle plus flexible de stockage. La mise à l'échelle de l'espace de stockage peut se faire dynamiquement en ajoutant des nœuds de stockage supplémentaires. De plus, il est possible de facilement répliquer les fichiers sur des machines physiquement distantes, ce qui évite la perte de données en cas de coupure de courant ou de catastrophe naturelle.

Dans ce qui suit, nous discutons les défis de conception des systèmes de stockage distribué les plus populaires. Puis, nous présentons les outils permettant d'étudier les performances de tels systèmes.

2.2.1 Conception des systèmes de stockage distribué

Plusieurs systèmes de stockage distribué ont été proposés pour répondre aux différents besoins des utilisateurs. Network File System (NFS) [12] est un protocole de stockage distribué développé initialement par Sun Microsystems [46]. Ce protocole offre la possibilité monter un système de fichiers à travers le réseau, ce qui permet aux utilisateurs d'accéder à des fichiers stockés sur des machines distantes. Cette solution est très pratique pour le partage de fichiers, mais pas pour le stockage de données à grande échelle.

OceanStore [47] et Farsite [48] sont des systèmes de stockage distribués qui proposent d'utiliser les machines des clients comme nœuds de stockage. Ces systèmes utilisent des mécanismes de sécurité, comme les protocoles Byzantins [49], pour protéger les données contre les clients malicieux. Néanmoins, ces systèmes sont lents et ne sont pas appropriés aux systèmes à haute performance.

Les systèmes de stockage distribué utilisent des techniques de synchronisation pour protéger les fichiers des accès simultanés. Par conséquent, l'accès à un même fichier par plusieurs clients peut causer une dégradation de performances. PVFS [50] et Vesta [51] ont réussi à améliorer le débit des accès parallèles en découpant les fichiers en petits morceaux distribués sur plusieurs machines. Cette technique permet aux clients d'accéder aux différentes parties du fichier simultanément, sans se bloquer mutuellement. PVFS et Vesta utilisent un serveur centralisé de métadonnées, ce qui peut causer des problèmes de performance avec des systèmes à grande échelle.

Afin d'éliminer la nécessité d'un serveur centralisé de métadonnées, GPFS [52] utilise une architecture entièrement distribuée, basée sur les *metanodes* pour la gestion des métadonnées. Le *metanode* est responsable de la mise à jour des métadonnées et utilise des verrous partagés pour synchroniser les accès simultanés.

zFS [53] et Lustre [54] utilisent le stockage objet [55] pour mieux supporter le passage à l'échelle. Les données et les métadonnées sont sauvegardées ensemble dans un seul objet, caractérisé par un identifiant unique. Ces systèmes permettent de stocker les objets sur des machines standards, ce qui réduit le coût du déploiement de la grappe de stockage. Ces systèmes offrent des mécanismes de réplication et de tolérance à l'erreur, ce qui permet d'assurer la disponibilité des données, même si un ou plusieurs nœuds de stockage tombent en panne. L'inconvénient de ces systèmes est l'utilisation d'une table d'allocation centralisée qui associe les données aux objets qui les contiennent. L'accès à cette de table peut devenir un goulot d'étranglement lors du passage à l'échelle.

Sorrento [56] propose un algorithme de hachage qui permet de trouver l'emplacement d'un

fichier sans passer par une table l'allocation centralisée. L'algorithme prend l'identifiant du segment de données à lire comme paramètre et retourne l'identifiant de la machine qui le gère. Cet algorithme ne tient pas compte de l'architecture physique de la grappe, ce qui peut causer des décisions de placement non idéales. Cette limitation a été corrigée dans Ceph [57] et GlusterFS [58].

Ceph est un système de stockage distribué qui incorpore la plupart des mécanismes modernes de stockage. Ceph utilise CRUSH (Controlled Replication Under Scalable Hashing), un algorithme de placement permettant de distribuer les données sur les nœuds de stockage OSD (Object Storage Daemon) d'une manière pseudo-aléatoire. L'administrateur système peut classifier les supports de stockage selon la vitesse, l'emplacement géographique, etc., pour améliorer les décisions de placement de l'algorithme. Par exemple, il est plus sécuritaire de stocker les réplicas d'un fichier dans des centres de données physiquement éloignés, pour éviter le risque de perte de donnée dans le cas d'une catastrophe naturelle. En plus du stockage de données, les OSD ont la responsabilité de maintenir la santé et le bon fonctionnement de la grappe. Si un nœud de stockage détecte qu'un de ses voisins n'est pas en marche, il diffuse l'information aux autres nœuds pour qu'ils prennent les mesures nécessaires [59].

Ceph permet d'installer plusieurs services sur la même grappe d'ordinateurs. Il offre par défaut un service de stockage d'objets, et au-dessus de ce service, il est possible de définir des périphériques de stockage, de stockage virtuel RBD (Rados Block Device), ou des systèmes de fichiers distribués CephFS (Ceph File System).

2.2.2 Performances des systèmes distribués

L'étude des performances d'un système distribué ne peut pas se faire en analysant chaque nœud individuellement. Les données collectées doivent être rassemblées et agrégées afin d'avoir une vue d'ensemble sur le réseau.

Netflow [60] est une technologie créée par Cisco permettant de surveiller le réseau en collectant les traces des différents périphériques Cisco, principalement les routeurs et les concentrateurs. L'architecture Netflow nécessite trois composants principaux : le traceur, le collecteur, et l'application d'analyse. Le traceur envoie périodiquement les données au collecteur pour qu'elles soient analysées. Les traces contiennent des informations sur le nombre de paquets traités, les adresses sources et destination, la taille des paquets, etc. Cette technologie est supportée seulement par les périphériques Cisco.

Simple Network Management Protocol (SNMP) [61] est un protocole de surveillance, similaire à Netflow, mais qui est supporté par presque tous les vendeurs. Il organise l'information

collectée dans une structure de données hiérarchique appelée *Management Information Base* (MIB). textitSNMP permet de récupérer des métriques de haut niveau comme l'utilisation CPU, le nombre de paquets perdus,etc.

Le traçage distribué souffre généralement d'un problème d'extensibilité. Il est difficile de collecter les traces de milliers de machines sans affecter les performances du réseau. Hussain et al. [62] ont discuté les exigences, la conception et les défis d'une infrastructure de traçage à grande échelle.

Pinpoint est un outil développé par Chen et al. qui permet de détecter les problèmes de performance des applications distribuées J2EE . L'idée est de marquer les requêtes des clients et de suivre leur cheminement dans le système. Cette méthode permet facilement de détecter le maillon faible d'une application distribuée.

X-Trace [63] est un outil similaire qui enregistre l'historique de chaque paquet réseau dans son en-tête. Contrairement à *Pinpoint*, *X-Trace* permet de générer des informations de très bas niveau, spécifiques à chacune des couches réseau. L'utilisation de X-Trace nécessite la modification de l'implémentation de la couche réseau dans le système d'exploitation.

Magpie [64] utilise l'instrumentation de l'application et du système d'exploitation pour étudier les performances des systèmes distribués. Les événements générés permettent de créer des relations de causalité, ce qui aide à détecter les goulots d'étranglement de l'application.

Dapper [65] est un outil de traçage distribué développé par Google. Il partage beaucoup d'aspects avec Magpie et X-Trace, mais il utilise une implémentation beaucoup plus évoluée, permettant de supporter des milliers de machines. Les auteurs ont défini trois exigences pour un système de traçage distribué efficace : faible surcoût, transparence et extensibilité.

2.2.3 Performance des systèmes de stockage distribué

Plusieurs travaux de recherche utilisent l'étalonnage pour évaluer la performance des systèmes de stockage distribué. Wang et al. [66] ont étudié la performance d'une grappe Ceph en mesurant l'impact du nombre des clients et des nœuds sur le débit de stockage. Les résultats ont montré que, en choisissant une configuration appropriée, il est possible d'atteindre 70% du débit maximal du matériel. L'utilisation du débit comme seule métrique de performance constitue une limitation de l'étude.

Une étude similaire a été faite par Van der Ster et al. [67] Les auteurs ont documenté leur expérience avec le déploiement de Ceph dans le département informatique du Conseil européen pour la recherche nucléaire (CERN). Les métriques de performance utilisées dans cette étude sont le débit, la consommation de mémoire, et le temps de rééquilibrage.

Plusieurs études utilisent l'étalonnage pour comparer les performances des systèmes de stockage distribué. Donvito et al. [68] ont utilisé IOzone [15], un outil d'étalonnage populaire, pour comparer le débit de HDFS, GlusterFS, et Ceph. Depardon et al. [69] ont comparé six systèmes de stockage distribué en termes de rapidité, accessibilité, et tolérances aux pannes.

Lee and al. [70] ont aussi utilisé l'étalonnage pour comparer plusieurs configurations de Ceph, notamment FileStore, KStore, et BlueStore. Les métriques de performance utilisées dans cette analyse sont principalement l'IOPS (en anglais, I/O Operations per second) et la latence moyenne des requêtes.

Gudu and al. [71] ont étudié l'impact de la journalisation sur la performance globale d'une grappe de stockage. Ils ont montré que l'utilisation d'un support de stockage rapide, comme la mémoire flash, pour garder le journal, permet d'améliorer considérablement la vitesse des opérations d'E/S. Ce résultat a été confirmé par la recherche menée par Poat and al. [72].

Les études présentées ci-dessus utilisent l'étalonnage pour évaluer les performances des systèmes de stockage distribué. Cette technique permet d'avoir une idée générale sur la santé de la grappe, mais ne fournit aucune vision sur le comportement du système et les goulots d'étranglement potentiels.

L'analyse dynamique par trace d'exécution est de plus en plus utilisée pour étudier le comportement des systèmes de stockage distribué. Zhang and al. ont proposé FSObserver pour analyser les performances de Ceph en se basant sur les traces PCAP [73]. Cet outil surveille le trafic réseau et extrait les paquets appartenant à Ceph. En analysant les en-têtes des paquets, il est possible de mesurer la quantité de données transférée par les nœuds de stockage. Cette technique est limitée aux échanges réseau et ne fournit aucune information sur les opérations d'E/S qui se produisent localement dans les nœuds de stockage.

Les versions récentes de Ceph sont instrumentées par Blkin [74], une librairie de traçage en mode utilisateur basée sur le format de Google Dapper [75]. Araujo et al. [76] ont utilisé cette instrumentation pour suivre une requête d'E/S à travers les différentes couches. La visualisation offerte par Zipkin permet de détecter les goulots d'étranglement.

Thereska et al. ont implémenté Stardust [77], un cadre d'applications permettant de suivre les requêtes d'E/S dans les systèmes de stockage distribué Ursa Minor [78]. Chaque requête est associée à un identifiant unique, ce qui permet de suivre la requête à travers les différents nœuds de stockage.

Les outils cités ci-dessus se basent principalement sur l'instrumentation en mode utilisateur des démons de stockage. Les données collectées permettent de mieux comprendre les latences des requêtes et de détecter plusieurs problèmes de performance. Cependant, le traçage en

mode utilisateur ne fournit aucune information sur les mécanismes du noyau impliqués dans les opérations de stockage, comme les ordonnanceurs de disques et les gestionnaires du réseau.

2.3 Stockage en mémoire

2.3.1 Gestion de la mémoire sous Linux

Les applications ont beaucoup évolué depuis la démocratisation de l'informatique. Leurs tailles ont beaucoup dépassé la capacité de la mémoire physique disponible. Pour résoudre se problème, les développeurs des systèmes d'exploitation ont proposé la notion de la mémoire virtuelle. Chaque processus est muni d'un espace d'adressage séparé qui lui donne l'illusion d'avoir toute la mémoire physique à sa disposition. Le chargement des données se fait d'une manière différée lorsque le processus en a vraiment besoin. Dans cette section, on discute les différents aspects et mécanismes permettant la gestion de mémoire dans le système d'exploitation Linux.

Mémoire physique

La mémoire physique est gérée par le système d'exploitation en termes de pages. Une page est la plus petite unité adressable par le MMU (Memory Management Unit) [5]. Sa taille est généralement entre 4K et 8K dans les architectures modernes. Le système d'exploitation utilise la structure de données *page* pour garder l'état de chaque page mémoire. Un compteur de référence *_refcount* est utilisé pour compter le nombre d'utilisateurs d'une page. Si *_refcount* est nul, la page est considérée comme libre.

Linux décompose la mémoire en trois zones principales :

- ZONE_DMA : Cette zone contient les pages directement accessibles par DMA. Ces pages sont utilisées pour les périphériques nécessitant un accès direct à la mémoire. ZONE_DMA32 a été proposée par Andi Kleen pour supporter les périphériques 32-bits.
- ZONE_NORMAL : Cette zone est utilisée pour les allocations normales du noyau. Elle est directement mappée dans l'espace d'adressage du noyau.
- ZONE_HIGH : Cette zone contient les pages qui ne sont pas directement mappées dans l'espace d'adressage du noyau. Ces pages sont mappées par le noyau au besoin à cause d'un manque d'adresses virtuelles.

Dans les systèmes 32-bits, l'espace d'adressage est de 4GB, 1GB de mémoire est réservé pour le noyau et 3GB pour les applications utilisateurs. Avec 1GB de mémoire virtuelle, le noyau ne peut faire un mappage direct avec une mémoire physique plus grande que 1GB. La mémoire

non mappée est considérée comme *high memory* et l'accès à cette mémoire nécessite un effort supplémentaire. Les pages doivent être chargées temporairement dans l'espace d'adressage du noyau en utilisant *kmap()* avant d'être utilisées. ZONE_HIGH n'existe pas dans les systèmes 64 bits, car l'espace d'adressage est suffisamment grand pour mapper toute la mémoire physique. La relation entre la mémoire physique et la mémoire virtuelle est schématisée dans la figure 2.8.

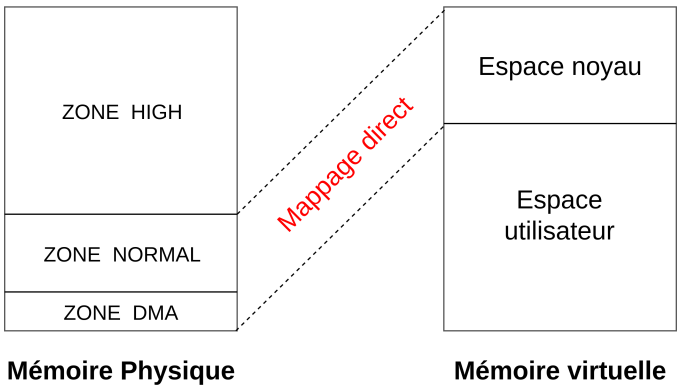


Figure 2.8 Mappage entre la mémoire virtuelle et la mémoire physique

Chaque zone de mémoire garde la liste de toutes les pages qui y sont associées. Cette liste permet de facilement localiser les pages libres pour les prochaines allocations. Si le nombre de pages libre d'une zone tombe au-dessous d'une certaine limite, le système d'exploitation utilise le mécanisme de réclamation de pages pour libérer l'espace mémoire. Le mécanisme de réclamation repose sur deux listes LRU (Least Recently Used) [79], une liste pour les pages active et une autre pour les pages inactives. Les drapeaux PG_active et PG_referenced sont utilisés pour contrôler le déplacement d'une page entre les deux listes. Une page doit être référencée deux fois pour être passée de la liste active à la liste inactive. Si la page n'est pas référencée pour une certaine période de temps, elle est passée dans l'autre sens (figure 2.9).

Le mécanisme a réclamation de pages se lance dans deux cas principaux :

- Lorsqu'une opération d'allocation de mémoire échoue. Ceci indique que la mémoire libre n'est plus suffisante pour servir les allocations.
- Périodiquement pour assurer que la mémoire de descend pas au-dessous d'une certaine limite. Cette limite est configurable dans *procfs*.

Espace d'adressage des processus

Chaque processus est muni de son propre espace d'adressage, défini par la structure de données *mm_struct* [79]. Cet espace d'adressage est formé de plusieurs segments, définis par

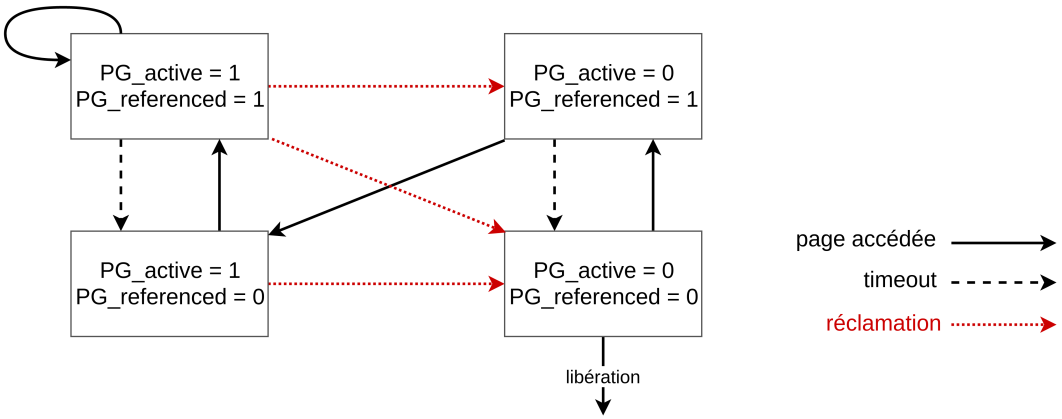


Figure 2.9 Mécanisme de réclamation de pages

vm_area_struct, qui représentent les régions de mémoire nécessaires pour l'exécution d'un processus comme le code du programme, la pile du processus et la mémoire tas (figure 2.10).

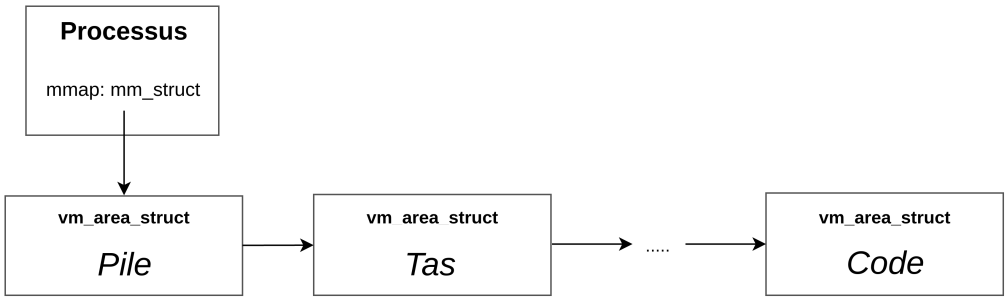


Figure 2.10 Espace d'adressage d'un processus

Les segments d'un processus donné peuvent être récupérés depuis le fichier */proc/$PID/maps*. Le système d'exploitation peut créer, supprimer, ou modifier les segments d'un processus en cours d'exécution. Par exemple, le noyau peut agrandir la mémoire tas à un processus si la mémoire donnée au processus n'est plus suffisante. Le noyau du système d'exploitation fait aussi partie de l'espace d'adressage des processus. Ceci rend l'exécution des routines noyau moins coûteuse puisqu'elle ne nécessite pas un changement de contexte à chaque appel système. Le noyau est protégé par des droits d'accès limités pour empêcher les processus utilisateurs d'y accéder.

Allocation dynamique de mémoire

Contrairement à l'allocation statique, l'allocation dynamique permet d'allouer et de libérer des objets dans la mémoire au fur et à mesure que le programme s'exécute. Le composant

principal permettant de gérer ce mécanisme est appelé *allocateur de mémoire*. Son rôle principal est de suivre l'utilisation de la mémoire du système au cours du temps afin de pouvoir servir les demandes d'allocations faites par des processus. Un allocateur de mémoire doit répondre à deux critères principaux. Il doit tout d'abord offrir un temps de réponse rapide en utilisant des algorithmes de recherche efficaces. De plus, il doit minimiser la perte de mémoire causée par le phénomène de fragmentation [80]. Ces deux critères sont en quelque sorte contradictoires : les algorithmes de recherche intensive qui retournent le meilleur emplacement dans la mémoire ont un temps de réponse long, et les algorithmes qui retournent le premier emplacement trouvé causent souvent une fragmentation sévère de la mémoire.

Plusieurs stratégies d'allocations existent dans la littérature :

- *Sequential Fit [81] :* Les blocs libres sont organisés dans une structure de données linéaire, généralement une liste doublement chaînée, et l'allocateur parcourt cette liste afin de trouver un espace libre approprié. Plusieurs variantes de cette stratégie existent. Best Fit retourne le plus petit bloc mémoire suffisant pour servir l'allocation. Cette méthode cause peu de fragmentation, mais par contre elle n'est pas efficace en termes de temps de réponse, car elle nécessite un parcours complet de la liste. First Fit retourne le premier bloc libre trouvé. Si le bloc est plus grand que la taille nécessaire, seulement la taille voulue est allouée et le reste est remis dans la liste des blocs libres. Cette technique fait que beaucoup de blocs de petite taille se localisent au début, et après un certain temps il n'est plus possible de satisfaire une opération rapidement, puisque les blocs libres de tailles suffisantes se trouvent à la fin de la liste. Pour résoudre ce problème, Next Fit fait la même chose que First Fit, mais commence la recherche à partir de la dernière allocation satisfaite.
- *Segregated Free Lists [82] :* Les blocs libres sont organisés dans un tableau de listes. Chaque liste contient les blocs libres de tailles égales. Les techniques de Sequential Fit peuvent être appliquées à chaque liste séparément.
- *Buddy Systems [83] [84] :* Ce mécanisme est un cas particulier de Segregated Free Lists qui impose des contraintes sur les tailles des blocs. Par exemple, Binary Buddy System impose des tailles de 2^n pour faciliter la recherche d'un bloc libre.

Wilson et al. [81] ont montré qu'il n'y a pas une stratégie d'allocation parfaite. La performance de l'allocateur dépend fortement de la charge du travail. Par conséquent, on trouve des allocateurs génériques qui donnent des performances acceptables avec la plupart des applications, et des allocateurs personnalisés pour une charge de travail spécifique.

Allocation en espace utilisateur L'allocation dynamique de la mémoire par les applications se fait par l'intermédiaire des librairies standards des langages de programmation.

Ces librairies standards offrent des fonctions comme *malloc()* et *free()* qui permettent aux applications d'allouer ou de libérer dynamiquement des objets en mémoire. Ces objets sont gardés dans la mémoire Tas. Si la mémoire Tas disponible n'est pas suffisante, l'allocateur demande plus de mémoire du système d'exploitation en utilisant l'appel système *sbrk()*.

Les allocateurs en espace utilisateur les plus connus sont Hoard [85], Ptmalloc [86], TCMalloc [87], and Jemalloc [88].

Allocation en espace noyau Le noyau offre aussi des facilités pour la gestion dynamique de la mémoire physique. *Kmalloc* permet de réserver une mémoire physique contiguë d'une certaine taille et *Kfree* permet de la libérer. Plusieurs drapeaux peuvent être utilisés pour spécifier les caractéristiques de l'allocation. Par exemple, GFP_ATOMIC est utilisé pour les opérations atomiques n'ayant pas le droit de se bloquer, comme celles utilisées dans les gestionnaires d'interruptions. La réservation d'une mémoire physique contiguë devient de plus en plus difficile au fil du temps à cause de la fragmentation. *Vmalloc* permet d'allouer une mémoire virtuellement contiguë, mais physiquement éparpillée. Une plage mémoire contiguë est seulement nécessaire dans le cas d'une interaction avec les périphériques, car ils ne supportent pas la notion de mémoire virtuelle.

Le noyau Linux utilise très fréquemment les opérations d'allocation et de libération. Les processus, les tampons d'E/S, et les paquets réseau sont tous représentés par des objets en mémoire. Pour minimiser le surcoût, il utilise un mécanisme de recyclage d'objets nommé *SLAB allocator* [89]. Les objets libérés sont gardés en mémoire et ils sont utilisés pour les prochaines allocations de même type. SLUB [90] offre un meilleur support aux architectures NUMA (Non-uniform Memory Access). SLOB est une version allégée utilisée dans les systèmes embarqués [89].

Gestion de cache

Les opérations d'E/S sont des opérations très lentes. Une opération de lecture de disque, par exemple, dure généralement une dizaine de millisecondes, ce qui représente une éternité en termes de cycles de processeur. Pour réduire le nombre d'opérations de disque, Linux garde une copie des fichiers récemment utilisés en mémoire principale dans une structure qui s'appelle *cache de pages*. Le cache se repose sur le principe de la localité temporelle. Si une donnée est utilisée à un instant t, il y a une très grande chance qu'elle soit réutilisée dans un futur proche. Lors d'une opération de lecture, les pages mémoires utilisées comme tampons d'E/S sont enregistrées dans le cache des pages. Les processus qui demandent la même information dans le futur seront servis directement à partir de la mémoire RAM, ce

qui améliore énormément le temps de réponse et évite le blocage des processus. Lors d'une opération d'écriture, le processus écrit les données dans le cache et continue son exécution. Ces pages sont après écrites sur le disque d'une manière asynchrone, à l'aide d'un mécanisme d'éviction expliqué plus en détail dans les prochains paragraphes.

Le cache de pages est défini par la structure de données *address_space* contenue dans l'*inode* du fichier propriétaire. Une page incluse dans le cache peut avoir un seul propriétaire. La liaison entre une page et son propriétaire se fait à l'aide du pointeur *mapping* défini dans la structure *page* (figure 2.11). *address_space_operations* définit les opérations d'E/S sur les pages, par exemple *readpage*, *writepage*, et *set_page_dirty*. C'est cette structure qui fait la liaison entre le cache et le système de fichiers.

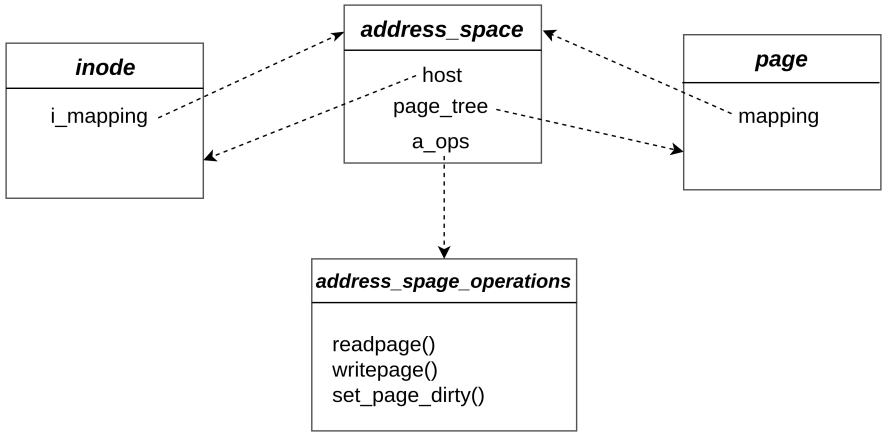


Figure 2.11 Relations entre les structures de données *page*, *inode*, et *address_space*

Le choix de la structure de données du cache est une décision très critique. Toutes les opérations d'E/S doivent passer par le cache, qui est généralement très grand. La recherche d'une page dans le cache peut introduire une grande pénalité si la structure de données utilisée n'est pas optimisée. Les développeurs du système d'exploitation ont utilisé un arbre binaire qui s'appelle *Radix Tree*. En utilisant l'index de la page mémoire, cet arbre est capable d'effectuer une recherche très rapide pour savoir si cette page existe déjà dans le cache ou pas.

2.3.2 Suivi de l'utilisation de la mémoire

Un des premiers travaux faits sur la visualisation de l'évolution de l'état de la mémoire au cours du temps est celui de Griswold et al. [91] Dans cet article, les auteurs ont instrumenté les opérations d'allocation et de libération de la mémoire du langage de programmation *Icon*, ainsi que l'activité du ramasse-miettes. L'instrumentation est faite en utilisant des *macros*, ce qui n'est pas sécurisé avec plusieurs files d'exécution. La trace générée est utilisée pour

créer une vue 2D. La mémoire est représentée par plusieurs bandes juxtaposées et chaque objet alloué est représenté par un rectangle dont la largeur est proportionnelle à sa taille.

GCspy est une plateforme de visualisation de la mémoire Tas proposée par Printezis et al. [92] La plateforme est développée d'une façon générique et modulaire, lui permettant de fonctionner avec n'importe quel gestionnaire de mémoire. L'architecture proposée est une architecture client-serveur. La collecte des données se fait au niveau du serveur et l'analyse se fait au niveau du client. Le serveur peut se connecter à un système en cours de fonctionnement sans le perturber. De plus, la plateforme offre un système de visualisation avancé, capable de s'adapter aux mémoires de grande taille.

Cheadle et al. [93] ont étendu GCspy pour supporter *dlmalloc*, l'allocateur de mémoire en espace utilisateur utilisé par défaut dans plusieurs versions du système d'exploitation Linux. Plusieurs améliorations ont été proposées pour mieux supporter les événements à haute fréquence, en diminuant la communication entre le serveur et le client. Il est aussi possible d'avoir la pile d'appel lors des événements générés. Les auteurs ont aussi défini des déclencheurs pour détecter des patrons problématiques.

GCspy montre l'évolution du contenu de la mémoire au cours du temps en offrant plusieurs images successives sous forme d'animation. Cette technique n'est pas appropriée pour des événements de haute fréquence. Pour résoudre ce problème, Moreta et al. [94] ont proposé une nouvelle vue 2D dans laquelle le temps d'un événement et l'adresse mémoire correspondent à l'axe des x et des y, respectivement. Ainsi, il est possible de voir l'état de la mémoire à n'importe quel moment dans une seule vue. Les blocs de mémoire occupés sont coloriés selon le processus qui a fait l'allocation, permettant de voir les processus les plus gourmands en mémoire.

VmpirTrace [95] est un outil d'analyse de performance qui offre un module pour suivre l'utilisation de la mémoire par les processus. Comme les outils présentés ci-dessus, VampirTrace se base sur l'instrumentation de l'allocateur de mémoire pour la collecte des données. La quantité de mémoire allouée virtuellement et le nombre de pages physiques utilisés instantanément sont donnés sous forme de courbes. La particularité de VampirTrace est le fait qu'il utilise aussi les compteurs matériels, par exemple le nombre de défauts de cache, pour donner plus de détails sur la micro architecture de la mémoire.

Reiss [96] a développé un outil permettant de montrer les différents objets alloués par un programme Java dans la mémoire Tas. Les relations entre les objets Java forment un graphe de correspondance : chaque objet est associé à plusieurs autres objets dont il est responsable. En fixant des hypothèses sur les types de relations entre les classes Java, les auteurs ont pu représenter ce graphe sous forme d'arbre facile à interpréter visuellement.

Memory Trace Visualizer (*MTV*) [97] permet de visualiser la séquence des accès mémoire d'un programme en cours d'exécution. La collecte des données se fait par l'intermédiaire d'une instrumentation dynamique du code source. L'outil de visualisation donne la carte des accès mémoire, ainsi que les lignes de code correspondantes, permettant aux programmeurs de trouver facilement l'origine des comportements problématiques. Les lectures et les écritures sont dessinées avec différentes couleurs. *MTV* permet aussi d'intégrer un simulateur de cache pour indiquer les moments exacts des défauts de cache.

2.3.3 Détection de la fragmentation de la mémoire

Le problème de fragmentation dépend de deux facteurs principaux : la stratégie d'allocation et la charge de travail appliquée. Ce problème peut être étudié en utilisant des techniques de simulation ou de traçage.

Randell [98] a étudié la fragmentation de la mémoire en utilisant un simulateur. Il analyse trois stratégies d'allocation, notamment *Best-Fit*, *Random* et *Reallocate*. Les résultats montrent que la fragmentation interne de la mémoire cause une perte considérable de l'utilisation de la mémoire. En se basant sur cette observation, les auteurs conseillent d'utiliser les techniques de segmentation, et non pas de pagination, pour améliorer le taux d'utilisation de la mémoire.

Shore et al. [99] ont utilisé un simulateur pour comparer les performances de *First-Fit* et de *Best-Fit* en termes de fragmentation externe. Ils établissent une relation entre la distribution des tailles des allocations et la performance de l'allocateur. *First-Fit* surpasse *Best-Fit* dans le cas où le coefficient de variance de la distribution est supérieur à l'unité.

Johnstone et al. [80] sont les premiers qui ont fait une étude qui mesure la fragmentation de la mémoire en utilisant le traçage. Quatre métriques de fragmentation sont proposées. Ces métriques se basent sur des comparaisons entre la mémoire utilisée au cours du temps et la mémoire totale demandée par l'allocateur. La mémoire utilisée est mesurée en traçant les appels *malloc* et *free*, et la mémoire demandée est donnée par les appels systèmes *sbrk*. Les tests effectués sur huit différentes applications montrent qu'avec une bonne technique d'allocation le problème de fragmentation est presque inexistant.

Bohra et al. [100] ont analysé la fragmentation causée par *Hummingbird* et *GNU Emacs*, deux applications qui génèrent un très grand nombre d'allocations dynamiques. L'analyse a été faite en deux phases. On crée une trace de toutes les allocations dynamiques faites par les applications, puis on exécute cette trace dans un simulateur qui modélise le comportement de plusieurs gestionnaires de mémoire. La métrique de fragmentation utilisée est la même que celle utilisée dans [80] : la taille des objets utilisés divisée par la taille de la mémoire tas.

Les applications en réseau ont un comportement imprévisible en termes d'allocation dynamique. Les paquets arrivent généralement en rafale et il est impossible de prévoir à quel instant un nouveau paquet va arriver. Les allocateurs offerts par les systèmes d'exploitation sont génériques et donnent une mauvaise performance avec ce type d'applications. Mamagkakis et al. [101] ont proposé une nouvelle implémentation personnalisée aux applications réseau qui fusionne les blocs libres adjacents après chaque libération de mémoire.

Les stratégies d'allocation sont souvent configurables, mais le choix des paramètres optimaux pour un type spécifique d'applications reste une décision difficile. Dans l'article [82], Del Rosso et al. ont proposé l'utilisation de l'algorithme génétique pour optimiser le choix des paramètres de la stratégie *segregated free lists*. La population initiale est générée en se basant sur le traçage des opérations d'allocation et de libération de mémoire.

2.3.4 Détection des fuites de mémoire

Une fuite de mémoire se produit lorsqu'un objet est alloué et n'est jamais libéré. Les fuites de mémoire constituent un problème très dangereux, que ce soit en espace utilisateur ou noyau. La fuite de mémoire en espace utilisateur cause un élargissement de la mémoire Tas des applications, ce qui peut mettre le système dans un état instable si la mémoire demandée par l'application est plus grande que la mémoire physique disponible. La mémoire allouée ne peut être libérée que lorsque l'application se termine. Les fuites de mémoire en espace noyau sont encore plus dangereuses. La mémoire non libérée reste allouée à jamais, jusqu'à l'arrêt de la machine.

La détection des fuites de mémoire est une tâche très difficile. Il est impossible de déterminer à un instant t si un objet est une fuite ou pas. On ne peut pas prédire s' il va être perdu dans l'avenir ou pas. Hauswirth et al. [102] ont développé SWAT, un outil de détection de fuites de mémoire qui se base sur le traçage des événements d'allocation et de libération des objets dans la mémoire Tas. Un objet est considéré une *fuite* s'il n'est pas accédé depuis longtemps. Cette méthode ne permet pas de détecter les fuites de mémoire d'une manière certaine, mais elle donne des indications sur les variables qui risquent d'être des *fuites*.

Pauw et al. [103] ont utilisé une approche similaire qui consiste à instrumenter la machine virtuelle de Java et à montrer tous les objets référencés dans la mémoire Tas. La détection des fuites de mémoire se fait en se basant sur des informations fournies par le programmeur sur la durée de vie des objets. Cette méthode est plus précise que celle utilisée dans [102], mais elle n'est pas pratique puisqu'elle nécessite l'intervention du programmeur de l'application.

Pour diminuer le surdébit de l'analyse de mémoire, Tsai et al. [104] utilisent deux phases de

détection. La première phase consiste à utiliser un modèle statistique de détection de patrons. Si ce modèle détecte une fuite potentielle, on passe à une deuxième phase où on utilise une instrumentation dynamique de la mémoire pour confirmer la fuite.

Carrozza et al. [105] ont utilisé Memcheck, un module de Valgrind [106], pour détecter des problèmes de fuite de mémoire dans les *Off-The-Shelf* (OTS) utilisés dans les intergiciels. Memcheck utilise l'instrumentation binaire dynamique pour surveiller les allocations et les accès mémoire.

Matias et al. [107] [108] ont utilisé le traçage pour détecter les fuites de mémoire dans l'espace noyau. Ils instrumentent les fonctions cibles en utilisant des *Kprobe*. Les auteurs ont confirmé une fuite mémoire connue dans le noyau en instrumentant les fonctions reliées à la création et à la libération des paquets réseau : skb_alloc(), skb_clone() et kfree_skb(). L'approche présentée n'est pas une approche générique, puisqu'elle ne permet pas de détecter les fuites de mémoire inconnues.

KEDR [109] est un outil d'analyse dynamique des modules noyau. Il permet de détecter les fuites de mémoire en interceptant les appels aux fonctions de gestion de mémoire dans le noyau. Une première étape consiste à instrumenter le module cible en utilisant les techniques de manipulation binaire. Lorsque le module se charge, KEDR analyse les fonctions appelées par le module afin de détecter les objets non libérés.

2.3.5 Gestion automatique de la mémoire et ramasse-miettes

Dans plusieurs langages de programmation, l'allocation et la libération des objets doivent être faites explicitement par le programmeur. Cette tâche complique le travail des programmeurs et peut causer plusieurs problèmes en cas d'erreur, notamment les fuites et la fragmentation de la mémoire. Pour éviter ces types de problèmes, des mécanismes de gestion automatique de la mémoire ont été proposés. Le ramasse-miettes est le composant principal permettant de gérer l'allocation et la libération des objets.

Le ramasse-miettes joue le rôle d'allocateur de mémoire. Il implémente plusieurs algorithmes permettant de trouver efficacement un espace libre pour les demandes d'allocation. Ces algorithmes de placement doivent être capables de rapidement placer les objets dans la mémoire, sans pour autant causer un problème de fragmentation. Les algorithmes d'allocation les plus populaires sont décrits en détail dans le paragraphe 2.3.1.

D'autre part, le ramasse-miettes doit nettoyer la mémoire en libérant les objets non utilisés. Certaines implémentations utilisent des algorithmes de comptages de références pour détecter les objets non accessibles. Ces algorithmes souffrent de certaines limitations, par exemple la

difficulté de détection des structures cycliques. De plus, garder un compteur pour chaque objet ajoute un grand surcoût en termes de mémoire. Pour résoudre ces problèmes, d'autres algorithmes de parcours de graphes ont été proposés. Ces algorithmes traversent le graphe des objets à partir de la racine et marquent les objets non accessibles en utilisant des bits de coloration. Le cycle de ramassage des objets est déclenché lorsqu'une certaine condition est remplie, par exemple, la mémoire disponible est inférieure à une certaine limite ou l'échec une demande d'allocation.

Plusieurs techniques ont été proposés pour améliorer les performances des ramasse-miettes. Les ramasse-miettes parallèles utilisent plusieurs fils d'exécution pour libérer les objets. Le cycle de ramassage peut se faire en parallèle avec l'exécution du programme, mais ceci n'est pas possible dans tous les cas. La machine virtuelle doit obligatoirement être arrêtée pendant une certaine période, pour s'assurer que les références ne sont pas modifiées par le programme au moment du nettoyage, cette période est appelée cycle de nettoyage *stop-the-world*.

Les ramasse-miettes peuvent aussi utiliser la technique de *compactage*. Après chaque cycle de nettoyage, les objets sont déplacés vers une région contiguë de la mémoire. Cette technique a deux avantages principaux. Premièrement, ceci permet d'éviter le problème de fragmentation, puisque les trous sont automatiquement éliminés après la libération des objets. Deuxièmement, l'allocation des objets est très rapide, car l'espace libre est connu à l'avance, aucun algorithme de placement n'est nécessaire.

Les ramasse-miettes générationnels divisent l'espace mémoire en plusieurs régions. Chaque région contient les objets de même âge. La plupart des ramasses-miettes générationnels utilisent deux générations : *Young generation* et *Old generation*. Un objet est d'abord alloué dans la *Young generation*. S'il survit après un certain nombre de ramassages, il est déplacé vers la *Old generation*. Cette séparation offre une grande flexibilité, car elle permet d'utiliser différents algorithmes de ramassages pour chacune des générations.

Plusieurs travaux de recherche utilisent des outils étalonnage pour étudier l'impact du choix du ramasse-miettes sur la performance des applications. [110–112] Ces outils exécutent un large éventail d'applications et mesurent leurs temps d'exécutions avec différents types de ramasse-miettes. Les applications utilisées ont des caractéristiques variées, par exemple la fréquence des opérations d'allocation, la durée de vie des objets, etc. Lengauer and al [113] ont discuté les avantages et les inconvénients des différents outils d'étalonnage disponibles.

L'analyse dynamique par traces d'exécution est aussi une technique très utilisée pour l'étude des performances des ramasse-miettes. Les études [114–116] utilisent le traçage en mode utilisateur pour mesurer la durée de vie des objets avec différents ramasse-miettes. En instrumentant les fonctions d'allocation et de libération, il est possible de dénombrer les objets

alloués et de mesurer la durée de vie de chacun d'eux. Les données collectées ne permettent pas de détecter la partie du code source qui génère ces allocations, ce qui présente une grande limitation de ces études. *Elephant tracks* [117] a permis de partiellement résoudre ce problème en instrumentant les entrées/sorties des fonctions, mais cette technique cause un très grand surcoût et ne peut pas être utilisée dans des systèmes en production.

2.4 Traçage

Dans cette section, on présente les traceurs les plus utilisés sur le système d'exploitation Linux.

2.4.1 Strace

Strace [118] est un outil de traçage développé en 1991 pour faciliter le débogage des applications. Strace surveille les appels système et les signaux, deux événements qui se produisent à la frontière entre l'espace utilisateur et l'espace noyau. Les événements enregistrés permettent aux analystes de localiser les problèmes de performance, même dans des logiciels dont le code source n'est pas disponible. La sortie de Strace contient le nom de l'appel système, ses arguments et sa valeur de retour. Il est possible de filtrer les événements en utilisant l'option *-e* qui permet de limiter l'analyse à un composant précis du système. Par exemple, on peut analyser le système de fichiers en exécutant Strace avec l'argument *-e read,write* pour limiter le traçage aux opérations de lecture et d'écriture. Le traçage peut aussi cibler un processus précis en se basant sur son *PID*.

La simplicité de Strace a encouragé les développeurs à l'intégrer dans la plupart des systèmes d'exploitation, par exemple Linux, Solaris et FreeBSD. Toutefois, Strace a aussi plusieurs inconvénients. D'abord, la visibilité offerte par les appels système est limitée. Plusieurs problèmes de performance nécessitent des événements de plus bas niveau pour être détectés. En plus, le surcoût introduit par Strace est énorme. L'utilisation des appels systèmes *ptrace* et *process_vm_readv* pour tracer les événements rend l'exécution du programme cible extrêmement lente. Finalement, Strace n'est pas approprié pour le traçage des applications parallèles. La sortie de Strace est écrite dans un fichier protégé par des mécanismes d'exclusion mutuelle, ce qui augmente dramatiquement le surcoût du traçage.

2.4.2 DTrace

Introduit dans Solaris 10, DTrace [119] est un outil de traçage développé par Bryan Cantrill, Mike Shapiro, et Adam Leventhal.

Contrairement aux autres traceurs, DTrace utilise l'instrumentation dynamique pour la collecte des données. Les points de trace peuvent être insérés dynamiquement dans un programme en cours d'exécution sans préparation préalable. On n'a pas besoin de recompiler le programme avec des drapeaux supplémentaires pour pouvoir le tracer, tout peut se faire à la volée. L'instrumentation statique de DTrace est aussi très efficace, ce qui a encouragé les programmeurs à instrumenter leurs applications sans avoir de soucis sur la performance. Un point de trace inactif est tout simplement considéré par le programme comme une instruction NOP, qui n'a presque aucun effet sur la vitesse du programme.

DTrace offre l'avantage de pouvoir instrumenter toute la pile logicielle dans un seul outil. La détection des bogues est beaucoup plus efficace avec une trace qui couvre l'application, les appels système, et le noyau du système d'exploitation. Cependant, la trace générée est généralement énorme, surtout si plusieurs points de trace sont activés. Pour diminuer la taille de trace et l'effort de post-traitement, DTrace offre des mécanismes de filtrage et d'agrégation, ce qui permet de générer des statistiques et des métriques directement depuis l'espace noyau. L'utilisateur peut contrôler le traceur en utilisant *D*, un langage de programmation de haut niveau qui offre des opérateurs et des structures de données similaires à celles du langage C. DTrace utilise une vérification très stricte pour éviter les paniques noyau qui peuvent se produire à cause d'une erreur de programmation. Ceci rend DTrace un outil sûr à utilisé dans un système en production. DTrace souffre d'un surcoût relativement grand. L'instrumentation dynamique offre beaucoup d'avantages en termes de facilité d'utilisation, mais par contre elle est moins efficace en termes de performances si on la compare avec l'instrumentation statique. De plus, DTrace n'est pas très performant avec les applications parallèles, à cause des mécanismes de synchronisation utilisés lors de l'accès au tampon d'écriture.

2.4.3 SystemTap

SystemTap [120] est un outil de traçage qui supporte l'instrumentation statique et dynamique. Il a été conçu comme l'alternative GPL de DTrace, qui est publié sous la licence CDDL.

Un langage de script, similaire à celui offert par DTrace, est fourni pour aider l'utilisateur à définir ses besoins. L'utilisateur peut définir un *handler* particulier pour chaque événement et calculer des statistiques pendant la capture. Plusieurs types d'événements sont supportés : entrées et sorties des fonctions, appels système, expiration d'horloge, etc. L'exécution des scripts de SystemTap passe par plusieurs étapes. Le script est d'abord converti à un programme C pour être compilé après comme un module noyau. Ce module est ensuite chargé dans l'espace noyau et les sondes de traçage sont activées. Lorsque l'exécution atteint le point de trace, le *hundler* correspondant est lancé. SystemTap est aussi capable de tracer les

applications en mode utilisateur en se basant sur les points de trace dynamiques fournis par *uprobe*.

Le format utilisé par SystemTap pour écrire les événements est lourd, ce qui limite considérablement sa performance avec des traces volumineuses. De plus, tout comme DTrace, SystemTap donne de mauvaises performances avec les applications parallèles.

2.4.4 Lttng

Linux Trace Toolkit next generation (Lttng) [3] est un traceur libre à faible surcoût. Lttng supporte le traçage en mode noyau et en mode utilisateur. Le traçage noyau se fait à l'aide des *Kprobes* ou avec des points de trace statiques.

Contrairement aux autres outils de traçage, Lttng est conçu pour supporter le traçage des applications parallèles. Un tampon séparé est dédié à chaque processeur pour éviter l'utilisation des techniques de synchronisation lors de l'écriture de la trace. La libraire RCU (Read Copy Update), sans verrou, est utilisée pour éviter le surcoût des mécanismes de cohérence de cache dans les processeurs multicœurs. Les événements de trace sont écrits sous le format *Common Trace Format* (CTF) dans des tampons circulaires et ils sont consommés par un processus séparé. Pour limiter le surcoût au maximum, les développeurs de Lttng ont choisi d'écraser des événements lorsque les tampons sont pleins, au lieu de bloquer le système pour attendre leur consommation. Lttng permet d'enregistrer les événements UST (Userspace Tracing) avec les événements noyau dans la même trace, ce qui permet de relier les événements bas niveau au code source de l'application.

2.5 Lecture et visualisation des traces

2.5.1 Babeltrace

Afin de diminuer le surcoût du traçage, plusieurs traceurs proposent des formats optimisés pour l'écriture des traces. Par exemple, CTF [121] est un format binaire très compact, permettant d'écrire une grande quantité d'information en utilisant le minimum d'espace disque. Les traces CTF ne sont pas lisibles directement et doivent être converties au format textuel avant de pouvoir être interprétées à l'oeil par les utilisateurs.

Babeltrace [122] est l'outil de référence permettant de manipuler les traces CTF. Par défaut, il convertit la trace vers un format textuel et l'affiche sur l'écran. Il permet aussi de convertir les traces CTF vers d'autres formats binaires afin de pouvoir les lire par des visualisateurs de traces ne supportant pas CTF. La figure 2.12 montre un échantillon des informations

sauvegardées pour chaque événement de trace.

Timestamp	[16:46:25.336025994]
Delay	(+0.000018075)
Hostname	Computer-K55VD
Event name	sched_switch
CPU	{ cpu_id = 2 }
Context	{ tid = 0, pid = 0 }
Parameters	{ prev_comm = "swapper/2", next_comm = "Xorg", ...}

Figure 2.12 Format Babeltrace

Babeltrace offre aussi une interface de programmation appelée *Babeltrace Python Bindings* qui permet de lire et d'analyser les traces en écrivant de simples scripts Python qui automatisent l'analyse des traces. Les analyses écrites avec cette interface de programmation sont généralement simples, par exemple l'extraction des métriques de performance ou la génération des statistiques sur la répartition des événements de la trace.

2.5.2 TraceCompass

Les traces système sont généralement de très grande taille et difficiles à investiguer. Trace Compass [123] est un logiciel libre écrit en Java permettant d'analyser plusieurs types de traces tels que CTF, BTF, GDB et PCAP. Trace Compass offre des vues graphiques qui modélisent plusieurs aspects de la performance du système analysé. Ces vues sont synchronisées ensemble sur une même échelle de temps. L'utilisateur peut naviguer librement la trace et les vues sont automatiquement rafraîchies pour refléter l'état du système pendant l'intervalle de temps sélectionné.

Trace Compass se base sur une approche par états. Un état est caractérisé par une date de début, une date de fin, et une valeur. Un événement de trace peut causer un ou plusieurs changements d'état. Par exemple l'événement de l'ordonnanceur *sched_swich(thread1,thread2)* change l'état du fil d'exécution *thread1* d'*actif* à *bloqué* et vice versa pour *thread2*.

La structure de données utilisée pour garder l'état du système doit respecter deux exigences : efficacité et extensibilité. Premièrement, elle doit offrir un temps de réponse rapide qui permet de récupérer une information sans causer un grand délai. Deuxièmement, elle doit être capable

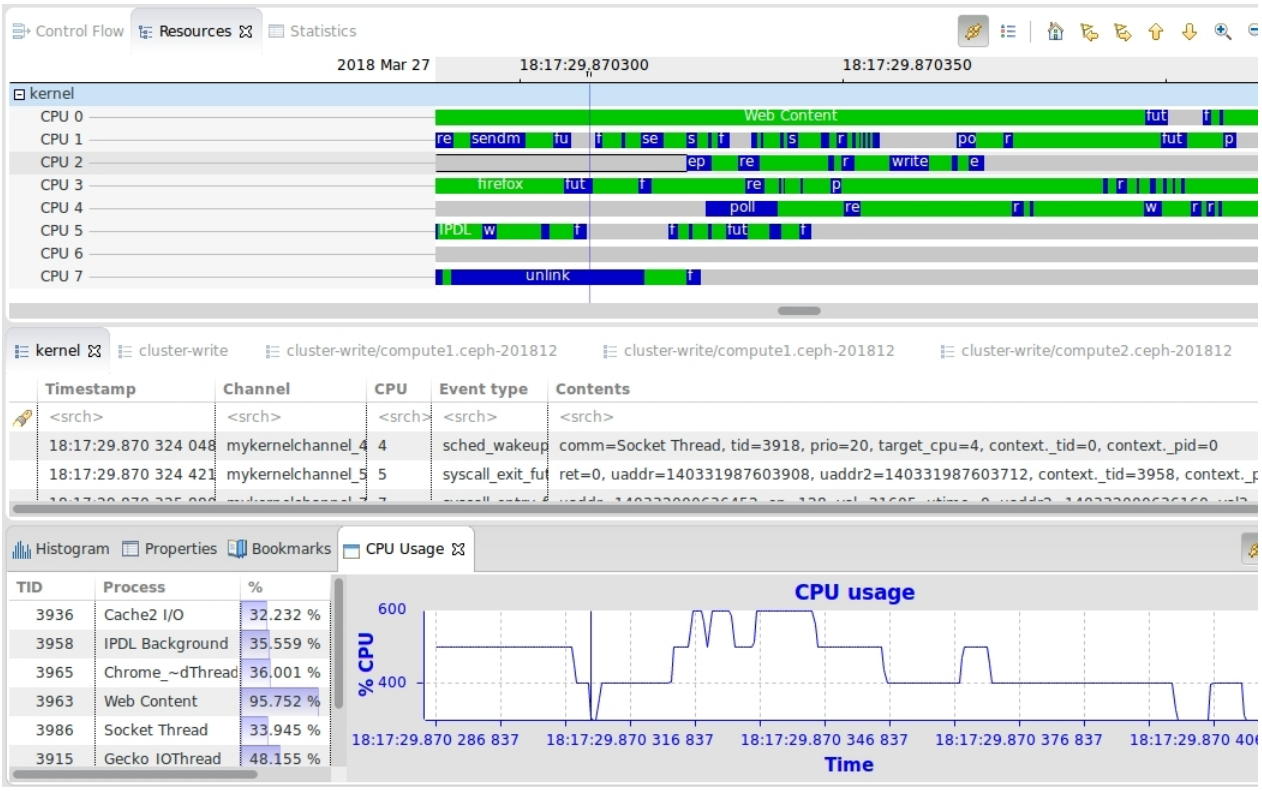


Figure 2.13 Visualisation d'une trace avec TraceCompass

de garder une grande quantité d'information vue la grande taille des traces qu'un système en production peut générer.

Les structures de donnée traditionnelles comme *segment-tree*, *interval-tree* respectent seulement la première contrainte. Ils offrent un délai de réponse rapide, mais ils ne peuvent pas supporter des traces de grandes tailles, car elles sont stockées dans la mémoire vive. Les bases de données relationnelles sont une autre alternative possible. Elles enregistrent les données sur le disque ce qui les rend extrêmement extensibles en termes de taille. Le problème est que le modèle de données utilisé par les bases de données n'est pas approprié à la gestion d'intervalles de temps.

TraceCompass utilise une structure de données appelée *Modeled State System* [124] qui organise les ressources du système sous forme d'arbre. Elle contient deux composants principaux : l'arbre des attributs et l'arbre des états. L'arbre des attributs est utilisé pour garder la structure du système sous forme hiérarchique. Chaque nœud dans l'arbre des attributs est associé à un nœud dans l'arbre des états. Cette structure de données peut être enregistrée sur la mémoire de masse, ce qui lui permet de supporter des traces de très grandes tailles. De plus,

les données sont triées selon les estampilles de temps, ce qui permet d'offrir un temps de réponse raisonnable.

2.6 Synthèse

Le stockage des données en mémoire et sur les périphériques d'E/S a un grand impact sur les performances des systèmes, ce qui motive beaucoup de travaux de recherche. Les approches basées sur l'étalonnage permettent de donner une évaluation de la performance globale du système, mais souffrent de plusieurs limitations. Premièrement, les charges de travail utilisées par les outils d'étalonnage ne reflètent pas le comportement réel des systèmes. Deuxièmement, les résultats de l'étalonnage peuvent être trompeurs à cause de quelques facteurs externes qu'on ne peut pas contrôler, comme la mémoire cache ou l'environnement système. Finalement, l'étalonnage fournit seulement une évaluation globale de la performance, mais n'aide pas à identifier les goulots d'étranglement et les anomalies difficiles à reproduire.

La revue de littérature démontre l'intérêt d'utiliser le mécanisme de traçage pour évaluer les performances des opérations de stockage. Cependant, les travaux existants souffrent de plusieurs défauts. Les mécanismes d'instrumentation utilisés sont généralement non optimaux et peuvent affecter le fonctionnement normal des systèmes. La plupart des études ne fournissent pas une évaluation précise du surcoût du traçage.

Les analyses existantes utilisent généralement une seule source de données et ne permettent pas de donner une vision globale sur le système. Rares sont les travaux qui incorporent les traces noyau et utilisateurs dans la même analyse. Il serait très utile d'utiliser un mécanisme de traçage standardisé qui permet de tracer plusieurs composants du système simultanément. La diversité des données permet de détecter des problèmes de performances difficilement identifiables.

La majorité des travaux existants proposent des outils qui automatisent le processus de lecture et d'analyse de traces, mais ils accordent peu d'importance à l'efficacité des algorithmes et des structures de données utilisés lors de l'analyse.

CHAPITRE 3 MÉTHODOLOGIE

3.1 Identification des besoins

3.1.1 Collecte des données

Une étude détaillée des différents traceurs existants sur Linux a été réalisée dans la Section 2.4. Chaque mécanisme de traçage vient avec des avantages et des inconvénients, que ce soit en termes de surcoût ou d'utilisabilité. Afin de respecter cette exigence de faible surcoût, nous avons choisi LTTng, vu sa conception adaptée aux systèmes à haute performance. Le nombre de points de trace utilisés dans les analyses doit être suffisamment grand pour fournir l'information nécessaire à l'analyse, sans pour autant générer un grand surcoût supplémentaire. Une bonne compréhension de la logique des applications et des mécanismes du système d'exploitation est nécessaire pour optimiser le choix des points de traces. Une autre caractéristique importante de LTTng est sa capacité de s'attacher simultanément sur des points de traces noyau et utilisateur, ce qui est indispensable pour nos analyses.

3.1.2 Mesure du surcoût du traçage

Le surcoût du traçage est parmi les enjeux importants dans notre recherche. Le surcoût peut se définir par l'augmentation du temps d'exécution du programme étudié. Ce type de surcoût est dit temporel. La taille des traces générées par le traceur est aussi considérée comme un surcoût de traçage. Un des buts de notre recherche est de pouvoir évaluer et diminuer le surcoût du traçage.

Puisque le surcoût est dépendant de la charge de travail, il faut analyser le système dans différentes conditions pour avoir une idée précise du surcoût de traçage. Pour attendre ce but, nous exécutons des outils d'étalonnages et nous comparons les résultats avec et sans traçage. Chaque test est répété plusieurs fois pour calculer le surcoût moyen du traçage. Des applications réelles sont aussi utilisées pour évaluer le surcoût du traçage dans des conditions réelles.

3.1.3 Corrélation des traces

Dans notre recherche, nous utilisons le traçage pour collecter des informations à partir de plusieurs composants simultanément. Nous définissons un canal de traçage pour chaque type d'événement, pour simplifier l'analyse par la suite. Si les traces sont collectées depuis la même

machine, la synchronisation des événements se fait en se basant sur l'horloge monotonique du noyau du système d'exploitation. LTTng récupère l'estampille de temps à partir du noyau Linux et l'écrit dans l'évènement de trace, ce qui permet de conserver l'ordre des événements, même s'ils sont générés sur différents coeurs du CPU.

Dans le cas d'un système distribué, il n'est pas possible de faire la synchronisation en se basant seulement sur l'estampille de temps, car chacune des machines a une horloge séparée. Nous utilisons l'algorithme *The fully incremental convex hull synchronization algorithm* [125] pour retrouver l'ordre partiel des événements. Cet algorithme se base sur la causalité sémantique entre deux événements appartenant à deux traces différentes. Les événements utilisés pour la synchronisation doivent respecter les conditions suivantes :

— L'événement x d'une trace cause l'événement y de l'autre trace
— Les événements x et y partagent un identifiant unique.

Par exemple, les événements réseau *send* et *receive* peuvent être utilisés pour la synchronisation, car l'événement *send* cause l'événement *receive* et ils partagent le même identifiant de paquet TCP.

3.1.4 Analyse des données

Le traçage permet d'avoir des informations précises sur les mécanismes de gestion de la mémoire et des périphériques d'E/S. Toutefois, les opérations de stockage peuvent générer des millions d'événements par seconde, ce qui rend l'analyse manuelle impossible. De plus, les événements contenus dans la trace sont de très bas niveau et difficiles à interpréter. Le développement des outils d'analyse automatique est donc une nécessité.

La manière la plus simple pour analyser une trace est la suivante. L'utilisateur choisit le type d'analyse et sélectionne l'intervalle de temps à analyser, l'outil lit séquentiellement les événements concernés et exécute l'algorithme de l'analyse. Le problème de cette technique est la nécessité de relire la trace à chaque fois que l'utilisateur sélectionne un nouvel intervalle de temps ou choisit une analyse différente.

Pour respecter l'exigence d'interactivité, nous avons opté pour une approche à état. Lors de la première lecture de la trace, une base de données est construite pour garder l'historique des états des composants du système. Cette base de données permet d'accélérer dramatiquement l'exécution des analyses par la suite, car elle élimine la nécessité de relire la trace pendant que l'utilisateur navigue à travers les intervalles de temps.

La base de données doit respecter deux exigences : efficacité et extensibilité. Elle doit en même temps offrir un temps de réponse rapide et pouvoir supporter des traces de grande taille.

Nous avons choisi MSS (*Modeled State System*) [124], une structure de données optimisée pour l'analyse des traces. MSS est composée d'un arbre des attributs qui garde la structure du système sous forme hiérarchique, et d'un arbre des états qui garde l'historique des états de chaque attribut. MSS peut être enregistrée sur la mémoire de masse, ce qui lui permet de supporter des traces de très grande taille. Un autre avantage de MSS est le fait que les données sont triées selon les estampilles de temps, ce qui permet d'offrir un temps de réponse rapide.

3.1.5 Abstraction des données

Les événements de trace sont généralement de très bas niveau et difficiles à comprendre par les utilisateurs. Il est donc essentiel d'utiliser des techniques d'abstraction permettant de représenter ces informations d'une manière simple et efficace.

Les métriques de performance fournissent une abstraction intéressante qui permet de représenter les performances du système sans avoir à examiner les traces manuellement. Dans notre recherche, Nous calculons les métriques de performance de stockage à partir des événements de trace. Une métrique peut être mesurée directement à partir des événements ou en se basant sur des métriques plus basiques. On peut donc définir une hiérarchie de métriques où chaque nœud est calculé à partir de ses fils.

La représentation graphique des données est aussi utile pour aider les utilisateurs à mieux analyser les traces. Nous fournissons un système de visualisation interactif contenant plusieurs vues qui couvrent tous les aspects nécessaires dans une étude d'analyse des performances de stockage. Les vues sont intégrées dans Tracecompass [123], un logiciel libre de visualisation de trace. Les vues offertes sont synchronisées sur la même base de temps. Lorsque l'utilisateur sélectionne un intervalle de temps, toutes les vues sont mises à jour pour correspondre à cet intervalle.

3.2 Environnement

3.2.1 Matériel

Les analyses et les tests de performance effectués ont été réalisés principalement sur une architecture Intel x86 SMP (en anglais, *Symmetric multiprocessing*). Le processeur a été utilisé avec une configuration de base. Les technologies avancées comme *Turbo Boost* et *Hyperthread* ont été désactivées afin d'avoir des résultats plus déterministes. Turbo Boost permet au processeur de changer automatiquement sa fréquence d'horloge en fonction de la

charge de travail, et Hyperthread permet à deux fils d'exécution de partager un même cœur physique.

Il est important de noter que les outils développés dans le cadre de cette thèse sont indépendants de l'architecture. La collecte de données se fait au niveau du système d'exploitation Linux et, par conséquent, les analyses peuvent être réalisées sur n'importe quelle architecture supportée par Linux.

3.2.2 Logiciel

Le système d'exploitation de référence utilisé dans notre recherche est Linux. L'utilisation d'un système d'exploitation libre et ouvert est une exigence du projet, car les analyses proposées nécessitent l'instrumentation du système. Le projet est difficilement réalisable sur des systèmes d'exploitation propriétaires.

Les outils développés dans le cadre de cette thèse sont des logiciels libres dont le code source est intégré dans LTTng, TraceCompass, ou disponible sur le référentiel Git de l'auteur [1]. Ceci permet aux utilisateurs de les tester et de proposer des extensions ou des corrections en cas de besoin.

3.3 Axes de recherche

Pour répondre aux objectifs de recherche, nous avons exploré deux thèmes principaux : le stockage de masse (Figure 3.1) et le stockage en mémoire (Figure 3.2).

En premier lieu, nous avons commencé par proposer une méthodologie d'analyse pour les périphériques de stockage de masse. Ensuite, nous avons étendu cette méthodologie pour prendre en compte les systèmes de stockage distribué.

En deuxième lieu, nous avons étudié les mécanismes de gestion de la mémoire en espace noyau et utilisateur.

Dans le reste de la section, nous fournissons une brève description des articles scientifiques issus de cette recherche. Ces articles apparaissent dans les chapitres subséquents et constituent l'essentiel de cette thèse.

Recovering Disk Storage Metrics from Low-level Trace Events Les périphériques par bloc jouent un rôle principal dans les systèmes informatiques. Ils sont utilisés principalement pour le stockage des fichiers, mais ils sont aussi nécessaires pour fonctionnement de

1. https ://github.com/houssemmh

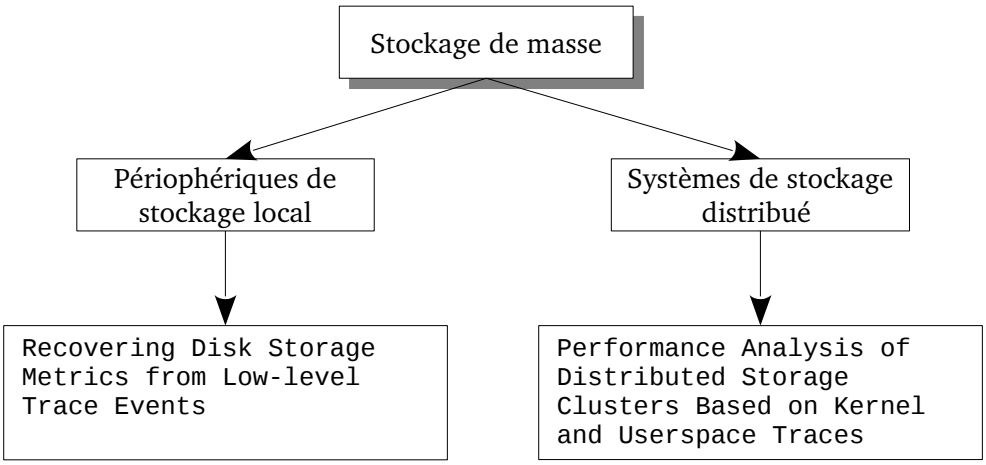


Figure 3.1 Les articles de recherche liés à la performance de stockage de masse

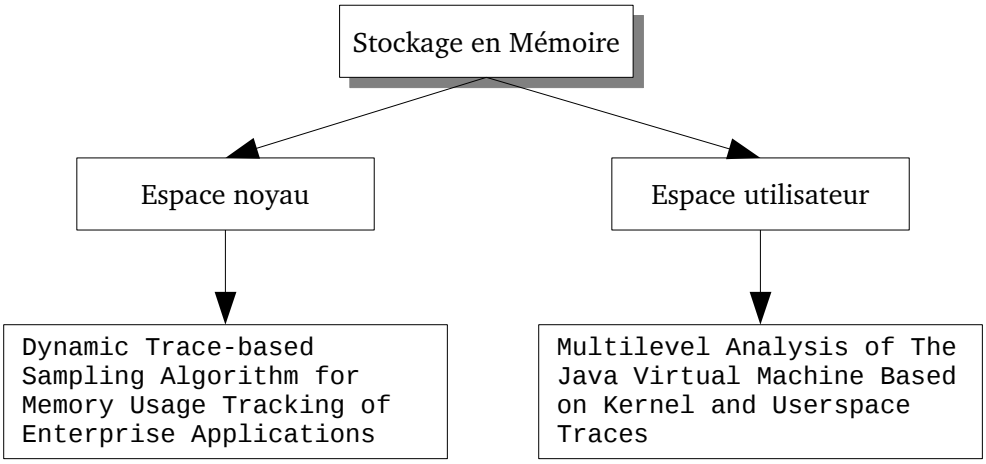


Figure 3.2 Les articles de recherche liés à la performance de stockage en mémoire

certains mécanismes importants du système d'exploitation, comme l'espace de pagination et la gestion des défauts de page.

L'analyse des performances des requêtes d'E/S du disque n'est pas une tâche simple, car la latence des requêtes dépend non seulement de la vitesse du disque, mais aussi du choix de l'ordonnanceur et de la charge de travail générée par les processus du système. Cet article fournit une étude détaillée de la performance des périphériques de stockage en utilisant le traçage noyau. Notre outil permet de suivre le cycle de vie d'une requête du disque, depuis sa création jusqu'à la fin de son traitement.

Les données collectées sont utilisées pour générer un historique d'états qui modélise la file d'attente de l'ordonnanceur du disque. Un des défis relevés par cet article est la génération des métriques de performance facilement interprétables à partir des événements ponctuels de la trace. Des structures de données sont proposées pour assurer un accès rapide à l'historique d'état lors de l'analyse.

Performance Analysis of Distributed Storage Clusters Based on Kernel and Userspace Traces Cet article aborde l'étude de performance des systèmes de stockage distribué. Dans cet article, nous avons étendu les analyses développées dans le premier article pour supporter les systèmes de stockage distribué. Nous avons utilisé l'instrumentation de Ceph, un système de stockage distribué populaire, pour étudier les mécanismes de stockage, de réplication et de reprise après panne. L'absence de base de temps commune entre les différentes machines constitue un défi majeur. Pour résoudre ce problème, nous avons utilisé un algorithme de synchronisation qui se base sur la causalité sémantique entre les événements.

Le deuxième défi relevé est le surcoût de traçage et la taille des traces produites. Nous avons proposé une approche de traçage dualistique basée sur deux sessions de traçage parallèles. La session légère active un nombre restreint de points de trace et enregistre les données temporairement dans la mémoire RAM. Cette session peut être toujours activée, car elle introduit un surcoût presque nul et ne consomme aucun espace sur le disque. Les événements de la session légère sont analysés à la volée pour détecter les anomalies. Lorsqu'un comportement étrange est détecté, la session détaillée enregistre des événements de très bas niveau sur le disque d'une machine centrale pour analyse approfondie. Cette technique nous permet de tracer des systèmes en production sans utiliser beaucoup de ressources supplémentaires.

Dynamic Trace-based Sampling Algorithm for Memory Usage Tracking of Enterprise Applications Le suivi de l'utilisation de la mémoire en utilisant le traçage présente un grand défi. Les événements de mémoire sont de très haute fréquence et l'enregistrement de tous les événements génère des traces gigantesques et difficiles à analyser.

Cet article propose un mécanisme d'échantillonnage dynamique permettant de faire des agrégations du côté noyau, afin de réduire le nombre d'événements générés. Pour chaque événement d'allocation ou de libération, le mécanisme fait des calculs pour voir si l'écart d'utilisation de la mémoire a dépassé un certain seuil et décider si un événement doit être généré. Ceci permet d'avoir une estimation assez précise de l'utilisation de mémoire sans introduire un grand surcoût.

Multilevel Analysis of The Java Virtual Machine Based on Kernel and Userspace Traces Les mécanismes de gestion automatique de la mémoire ont beaucoup facilité le travail des développeurs. Néanmoins, le comportement autonome du ramasse-miettes introduit un facteur d'imprévisibilité pour la performance des applications. Le but de cet article est d'étudier les mécanismes internes de la machine virtuelle Java.

Nous avons proposé une analyse multiniveau qui prend en compte plusieurs composants de la machine virtuelle Java : le ramasse-miettes, le compilateur JIT et le gestionnaire des fils d'exécution. Cette analyse permet de voir en détail ce qui se passe à l'intérieur de la machine virtuelle lors de l'exécution d'un programme. Ceci nous a permis d'évaluer l'impact de la gestion automatique de la mémoire sur le temps d'exécution d'un programme. La corrélation entre les événements noyau et utilisateur permet de détecter des aspects de performance invisibles aux autres outils existants. En combinant l'analyse de Java avec les analyses proposées dans les autres articles, nous sommes capables d'étudier des problèmes très intéressants, par exemple l'impact de l'ordonnanceur du disque sur le temps d'exécution d'une application Java.

CHAPITRE 4 ARTICLE 1 : RECOVERING DISK STORAGE METRICS FROM LOW-LEVEL TRACE EVENTS

Authors

Houssem Daoud and Michel Dagenais
Department of Computer and Software Engineering, Ecole Polytechnique de Montreal, Montreal, H3T 1J4, Canada

E-mail : {houssem.daoud, michel.dagenais}@polymtl.ca

Published in Software : Practice and Experience, 01 February 2018
https ://doi.org/10.1002/spe.2566

4.1 Abstract

Block devices, such as magnetic disks, are non volatile data storage devices that transfer data in fixed-size chunks. They are the main non-ivolatile memory that holds the file system and they are also used in virtual memory mechanisms like swapping and page fault handling. Investigating storage performance issues requires a full insight into the operating system internals. Kernel tracing offers an efficient mechanism to gather information about the storage subsystem at runtime. Still, tracing output is often huge and difficult to analyze manually.

In this paper, we introduce a framework to compute meaningful storage performance metrics from low-level trace events generated by Lttng. A stateful approach is used to model the state of the storage subsystem. Efficient data structures and algorithms are proposed to offer a reasonable response time, allowing the user to navigate throughout the trace and to retrieve metrics from any time range. The framework includes a visualization system that provides different graphical views that represent the collected information in a convenient way. These views are synchronized together, forming a comprehensive perspective that makes storage performance investigation a much more comfortable task. Different use cases are presented to show the usefulness of the framework in real world applications.

4.2 Introduction

Operating systems rely on secondary storage devices to provide a more effective multiprogramming support. When an application's data is bigger than the available physical memory,

some memory pages are saved to the block device and they are reloaded by the page fault handler when needed. In addition, disk drives are used to hold the filesystem, in which all files are saved in a tree-like hierarchy. Every file I/O operation performed by user applications, or by the operating system itself, is handled by the storage subsystem. Recent processors provide many techniques to hide memory latencies such as deep pipelines, speculative execution and multithreading. These methods allow the processor to handle more operations while the data is being retrieved. Still, not all latencies can be hidden, especially those involving secondary storage operations.

To narrow the gap between processors and disks performance, operating system developers have introduced many optimizations like caching [5], scheduling [126], and prefetching [8]. Caching keeps file contents in main memory so that it can be quickly retrieved in the future. Prefetching retrieves data from storage into cache before being explicitly requested by applications, based on access prediction techniques. Disk scheduling is keeping I/O requests in a waiting queue and merging and reordering them in a way that reduces the response time and increases the disk bandwidth.

Despite their benefits, these optimizations have complexified the storage subsystem architecture and, as a result, it is very difficult to pinpoint the origin of a storage-related problem : a long access latency can be caused by the filesystem, the scheduler, the driver, or by the disk itself. When it comes to disk performance evaluation, two main techniques can be used : Benchmarking [20] and Tracing [127]. Benchmarking is the process of running a defined workload, or a set of workloads, in order to estimate the performance of the system. The first limitation of benchmarking is that the workloads used are generally not representative. Real world applications are so diverse that they cannot be simulated by synthetic workloads. In addition, benchmarking is affected by the system configuration. In large memory systems, benchmarking results can be biased due to the effect of cache [2]. Furthermore, benchmarking is only useful for performance evaluation, and not for debugging. It does not provide useful information that helps in investigating storage performance issues. Tracing offers very detailed behavioral data about the storage subsystem. For example, we can know exactly when an I/O request is created, how the scheduler dealt with it, and how much time the disk drive took to process it. It is important to make sure that the overhead introduced by tracing doesn't have a noticeable impact on the normal behavior of the system as this is not acceptable in production systems.

In this paper, we propose an advanced tracing-based framework that helps programmers and system administrators in debugging storage-related issues. The proposed solution targets the Linux operating system, but it can be adapted for other operating systems. We have defined

the following goals for the storage performance analysis framework :

G1 *Low-overhead tracing :* We instrumented the operating system and extended Lttng, a low-overhead tracer, to provide information about the storage subsystem.

G2 *An interactive analysis :* The amount of data generated by the tracer is tremendous and needs a serious post-processing effort to be analyzed. We used an efficient tree-based data structure that keeps the state of the system in order to ensure a quick analysis time.

G3 *A comprehensive visualization system :* We defined a set of relevant metrics and implemented a visualization system that covers all the aspects necessary to debug a storage-related issue.

The rest of the paper is organized as follows : Firstly, after discussing important background concepts, we present the architecture of our framework and we show the role of each component. Secondly, we explain how disk storage metrics can be computed from low-level trace events. Then, we present different use cases to highlight the use of the proposed framework. Finally, we evaluate the efficiency of the framework and we conclude by discussing possible enhancements.

4.3 Background

In this section, we describe the architecture of the storage subsystem and we discuss the important storage performance metrics. Then, we present the different tracers available. Finally, we present a selection of storage analysis tools.

4.3.1 Storage Subsystem Architecture

The storage subsystem in the Linux operating system is very modular. Many components are interacting together to offer a good I/O performance to userspace applications (Figure 4.1).

Userspace applications are not allowed to interact directly with the hardware. The operating system offers a set of system calls for the applications to initiate an I/O operation such as open, close, read, write, etc. Many flags are offered to control the behavior of those calls. Each system call is linked to a function defined inside the Virtual File System Layer (VFS), which provides an abstraction of the file systems. For example, the read system call is linked to the function *vfs_read*. The VFS issues the call to the target file system which creates the I/O request and provides it to the disk scheduler. The request is then processed by the

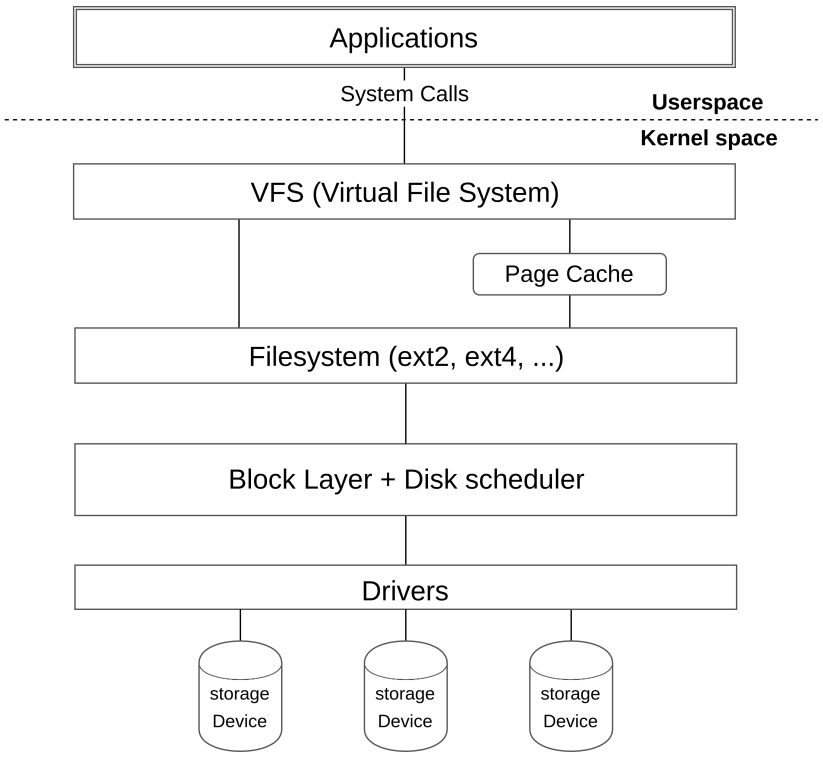


Figure 4.1 Storage Subsystem Architecture

device driver and an interrupt is generated when the processing is finished. An I/O request is defined as a linked list of BIOs (Block I/O), which gives the flexibility to define an I/O buffer in a scatter-gather way.

Two main queues are present in the storage subsystem : the elevator queue and the dispatch queue. The elevator queue is defined in the block layer and is managed by the I/O scheduler. It is where requests are reorganized and merged in a way that optimizes disk access. Many performance criteria can be used to evaluate the effectiveness of the scheduler namely disk throughput, the fairness between processes and the respect of real-time constraints. The most common disk scheduler available in Linux are *Noop*, *Deadline*, *Anticipatory*, and *CFQ* [6].

The dispatch queue is where the requests are kept in order to be processed by the disk when it is ready.

The page cache is another crucial component of the storage subsystem. The philosophy of the page cache reposes on temporal locality. A data used now is likely to be used again in the near future. A Radix tree [128] is used to hold the necessary data in an efficient way. Cache lookup is done based on the virtual address of the required memory page. The conversion between the virtual address and the physical address is hardware-assisted. It is important to mention that our work focuses on the performance of I/O requests that involve a disk

activity. The analysis of the efficiency of the page cache is out of the scope of this paper.

4.3.2 Storage Performance Metrics

Measurements are very important to quantify system performance. They are used to evaluate system efficiency and to detect strange behaviors. At the same time, metrics should be carefully chosen to get precise information and avoid skewing the results.

Storage performance can be evaluated at different levels, from the application performing I/O operations down to the disk controller. The most commonly used performance metric is the *throughput*, which is defined as the quantity of data a disk can process in a certain amount of time. Computing the throughput from userspace can provide misleading results. For example, a file reading operation does not involve any disk operation if the data is found in the cache. In such a case, the computed throughput does not reflect the read performance of the disk. Wrong results can also be provided if the userspace application is interfering with another background process performing I/O operations at the same time and, as a result, the throughput computed in userspace is lower than the real value. To get precise information, a better approach is to compute metrics based on data from the kernel space, where we have a full view of all the I/O activity in the system, including userspace processes and the kernel itself.

Response time is defined as the duration between the insertion and the completion of an I/O request. Providing a single metric for request latency is ambiguous : a high latency can be caused by different actors : the VFS, the filesystem, the block layer, and the disk driver. In a storage debugging task, it is very useful to have a detailed latency for each layer, making it easier to pinpoint the origin of the problem.

Disk utilization, defined as the fraction of time that the disk is processing I/O operations, is calculated based on request insertion and completion events. Reducing idle time by interleaving background disk activity with the normal workload has proved to enhance the overall performance of the system [129]. Schedulers are generally designed to keep the disk busy as much as possible [126].

Seek time is another widely-used metric to characterize the I/O workload. A low value indicates a sequential activity, whereas a high value indicates a random one. Random I/O is one of the biggest causes of performance degradation, because the movement of the disk head —in the case of a mechanical disk— is very slow. Even for SSD drives, sequential I/O is faster because it benefits from the cache and the prefetching mechanisms implemented in the disk drive. With the aim of minimizing disk seeks, many scheduling algorithms have been

proposed to sort requests based on their target sectors [130].

4.3.3 Kernel Tracing

Tracing provides details about system execution at runtime. Tracing can be done in userspace or in kernel space. Userspace tracing is useful to track the execution flow of applications. It is particularly useful with multi-threaded programs because the execution order is not deterministic and some unexpected situations can occur like deadlocks and starvation. MPE [131] is a software package that provides a tracing facility for MPI programs. Jumpshot [132] provides a graphical visualization for MPE traces to facilitate the detection of problems. TAU [133] is a similar tracing toolkit able to automatically instrument parallel programs written in different programming languages using a dynamic instrumentation mechanism. Those tracers are not useful in our work since they provide none or very little information about storage performance.

Kernel tracing allows to collect information from the operating system. It is possible to trace system calls, interrupts, function calls, etc. This mechanism is available in almost all modern operating systems such as Linux, Windows, Unix and Solaris. Since the main focus of this paper is Linux, we will present the tracers available on this platform, namely Strace, DTrace, SystemTap, Blktrace and LTTng.

Strace

Strace [118] is a tracing tool, developed in the early nineties, to help developers debug programs by examining their interaction with the operating system. It achieves this by monitoring system calls and signals, two types of events that happen at the user/kernel boundary. It is possible to filter the events using the *-e* option, which allows to focus the tracing towards a specific analysis. The simplicity of Strace has encouraged operating systems developers to support it in most Unix-like operating systems such as Linux, Solaris and FreeBSD. However, this simplicity can also be considered as a weakness of Strace. In fact, system calls offer a limited visibility into the system. Moreover, Strace comes with a massive performance overhead, especially with multithreaded applications, making the system remarkably slower than usual [134] [135].

DTrace

DTrace [119] is a tracing facility introduced in Solaris 10. Unlike other tracing tools, DTrace is able to perform dynamic instrumentation by inserting probes into a running software

without any previous preparation. Namely, there is no need to precompile the software with special flags or instruments in order to trace it, everything can be done on the fly. One of the biggest advantages of DTrace is the ability to trace the entire software stack, from the application itself to the deepest layers of the operating system, in one consistent tool. With the aim of reducing the post-processing effort, filtering and aggregation mechanisms are provided. Unfortunately, DTrace suffers from performance issues when tracing heavy workloads [136]. Dynamic instrumentation is very convenient to use, but it burdens the system with a noticeable overhead, compared to static instrumentation. In addition, the Linux version of DTrace is not well maintained.

SystemTap

SystemTap [120] is a tracing framework that supports static and dynamic instrumentation. A scripting language, similar to the one offered by DTrace, is provided to allow users to inspect system activity at runtime. Users can easily define events and their corresponding handlers. Many types of events are supported : function entry/exit, system call, timer expiration, etc. Executing a SystemTap script goes through different steps. The script is firstly translated to a C program and then compiled to a kernel module. The kernel module is then loaded to the kernel space and the probes are activated [137]. When an active probe is hit, the corresponding handler is executed. SystemTap suffers from a noticeable performance overhead caused by synchronization mechanisms when tracing heavily multithreaded applications [135].

Block Layer Tracing : blktrace

Before the 2.6 kernel series, the block layer architecture was very inflexible, causing serious performance issues. As a result, Linux Kernel developers have decided to re-implement this layer using more efficient algorithms and data structures. The new version is much more effective, but it is also more difficult to debug. Blktrace is a low-overhead I/O tracer developed by Jens Axboe, the current Linux Kernel maintainer of the block layer, to analyze the I/O performance of the system. Blktrace is designed with efficiency in mind : trace files are written in binary format, and a separate file is created for each processor. Data is generated using the static instrumentation mechanism. The major weakness of blktrace is that the visibility is limited to the block layer. The storage subsystem is very complex, and a consistent performance analysis tool needs to cover the whole I/O stack, including the system calls, the file systems and the device drivers.

LTTng : The Linux Trace Toolkit next generation

is a low-impact open source Linux tracing tool that supports kernel and user space tracing. Kernel tracing is done by hooking dynamically on Kprobes [138], or statically on tracepoints. LTTng achieves very high performance with mutlithreaded applications by allocating per-CPU output buffers and using lock-free data structures. The read-copy-update (RCU) mechanism is used to avoid cache coherence overhead in multi-core systems. Trace events are written in circular buffers and they are consumed by a separate process. To achieve a minimum overhead, LTTng designers chose to discard events rather than blocking the running application. Lttng-UST can also be used to add userspace tracepoints.

Performance

The tracer that we use in our tool must respect many criteria. Firstly, the tracing should not disturb the normal behavior of the system. Choosing a low-overhead tracer is crucial to make our tool usable in production system. Secondly, the tracer must also be able to collect and correlate events from different levels of the system.

The relatively high overhead introduced by DTrace and SystemTap and their lack of support on Linux makes them inconvenient for our work. Blktrace, on the other hand, offers good performance but can only collect events from the block layer, which are insufficient for our analysis.

The low-overhead of LTTng, compared to other options, makes it the best candidate to use in big data centers where storage debugging is highly needed. Furthermore, it is able to collect events from different layers of the operating system. We can target the block layer, the filesystem and the system calls at the same time. It is also possible to instrument a target application by adding userspace tracepoints, in case we want to know which component of the application generates more I/O workload. The collected events for the different layers are synchronized with the monotonic clock of the system and written to the same buffer.

For these reasons, LTTng 2.8 is used as the main tracing platform in this paper.

4.3.4 Storage analysis tools

Many storage performance monitoring tools are available on Linux such as *iotop*, *iostat* and *sysstat*. Those tools use *procfs* and *systfs* to retrieve kernel statistics. The information allows the monitoring of the I/O activity of processes and disks, but is not detailed enough to detect storage performance issues that may occur in one of the operating system components. In

this section, we introduce more advanced tools that use tracing to get more fine-grained information about the storage subsystem.

Seekwatcher

The amount of data generated by blktrace is tremendous and it is almost impossible to pinpoint performance issues by examining the trace file manually. Seekwatcher [39] was developed to visualize the output of blktrace by creating graphs representing different aspects of the storage subsystem. Time charts are used to represent three major performance metrics : Seeks per second, IOPS (IO operations per second), and disk throughput. A heatmap-based view is also provided to show the disk head position throughout the time.

The information provided by Seekwatcher is not enough to debug many storage related problems. It doesn't give any hint about problematic requests and the content of the waiting queues when a problem is detected. The information provided by our trace-analysis tool is much more precise and helps in pinpoint the origin of performance problems.

Oracle ZFS Storage Software

is a storage analysis framework provided with Oracle appliances. This tool uses DTrace to get low level data about disks and shows it as graphs in a web-based application. A wide range of views are offered to the user. Disk head movement is shown in a time chart, and a latency distribution is presented as a heatmap where the time is on the X-axis, the latency is on the Y-axis, and color shades are used to illustrate the number of I/O operations in a specific latency range.

Representing latency as a heatmap is useful for analysis purposes [43] (Figure 4.2). Classical methods of presentation, like using a graph of latency average, do not provide a clear idea about the latency distribution, since the outliers cannot be identified beyond an average. In a heatmap, in addition to the maximum and the minimum values, the average can be visually identified by finding where the darkest hues are grouped. ZFS Storage Software updates the views in real-time, as soon as new trace events are available, which is very useful for live monitoring. Despite the overhead introduced by DTrace, this tool has proved to be very useful and safe to use in production systems. Unfortunately, DTrace is basically developed for Solaris and is not well maintained for Linux, so it is not possible to use it on recent Linux versions.

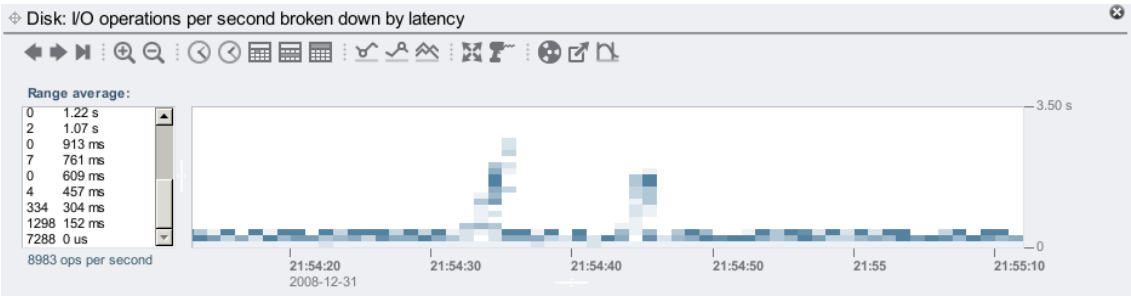


Figure 4.2 Latency Heatmap [43]

Windows Performance Analyzer (WPA)

is a performance monitoring tool included in the Windows Assessment and Deployment Kit [44]. With a huge variety of views, WPA covers almost all the metrics needed for storage performance analysis. Graphs are created based on ETW (Event Tracing for Windows) traces, produced by the Windows Performance Recorder (WPR). WPA provides low-level information about the storage subsystem. The user can get a detailed list of I/O requests processed in a selected time range and easily sort and filter them by defining some criteria. Many performance metrics are recovered from the trace, like IOPS, throughput, disk usage, number of seeks per second, etc. The visualization is mostly based on charts, where the X-axis is the time, and the Y-axis is the metric presented. The biggest limitation of WPA is being platform-dependent, making it unusable in other operating systems.

4.4 Architecture

The architecture of the proposed framework is shown in Figure 4.3. It consists of different components : the kernel tracer, the storage subsystem analyzer, and the visualization system. In this section, we present the implementation details of these components.

4.4.1 Kernel Tracer

Choosing a good set of tracepoints is essential to understand the internal mechanisms of the storage subsystem. The number of active tracepoints should be big enough to give a clear information about the storage subsystem, but not so big that tracing affects the normal behavior of the system. System call events are useful to see the interaction between userspace processes and the operating system, but the information they provide is not enough to understand the storage subsystem internals. For example, a read system call doesn't involve any disk activity if the data is available in the cache. To make the analysis more precise, our

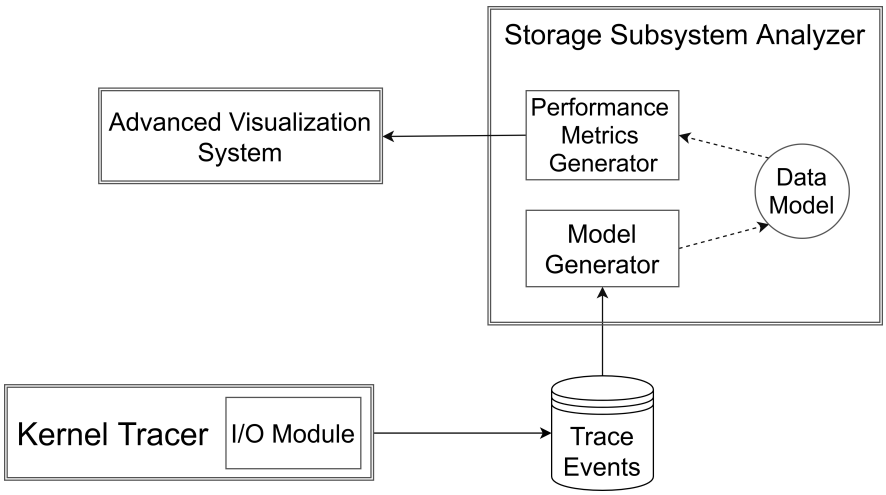


Figure 4.3 General Architecture of the System

framework also uses tracepoints from inside the block layer in addition to the system calls. The instrumentation of this layer gives us valuable information about I/O requests, waiting queues and the disk scheduler.

The Linux Kernel already contains interesting tracepoints related to the storage subsystem like request insertion, completion and failure. The set of tracepoints currently available in the mainline version of the Kernel is not enough to do an in-depth analysis able to track request merging and asynchronous requests. We added an I/O module to LTTng to provide the missing information. This module is based on Kprobes, a dynamic instrumentation technology allowing to add new tracepoints to the system at runtime. The module we developed provides three extra tracepoints. *block_rq_merge* is called when two requests are merged together. *lttng_statedump_block_device* is an event that provides the mapping between the block device names and IDs. It is very useful to have this matching since users generally identify the devices by their names, not their IDs. *mark_page_dirty_IO* is called when a memory page is marked as dirty and is used to know the processes that contributed to an asynchronous request.

The list of tracepoints required by our framework is presented in Table 4.1.

4.4.2 Storage Subsystem Analyzer

Stateful Analysis

Tracing efficiently provides useful information for the analysis of the storage subsystem. Execution traces contain very detailed information about the whole I/O stack, from the VFS

Table 4.1 Required tracepoints

Tracepoint	Description
block_getrq	This event indicates the creation of the I/O request data structure.
block_rq_insert	This event indicates the insertion of an I/O request into the elevator queue of a disk. Its payload holds the ID of the disk, the sector number, the size and the flags of the inserted request.
block_rq_issue	This event indicates that an I/O request is fetched by the device driver in order to be processed. At this point, the request migrates from the elevator queue to the dispatch queue.
block_rq_complete	This event indicates that the disk has finished handling the I/O request and the request is removed from the dispatch queue.
block_bio_backmerge	This event indicates that a new BIO (Block I/O) is added to an existing I/O request.
block_bio_frontmerge	
block_sleeprq	This event indicates that the I/O request is not inserted because the waiting queue is full.
block_rq_merge	When two requests are targeting adjacent sectors, the scheduler merges them together to accelerate the processing. This event helps us track the merged requests by holding their IDs in the payload.
lttng_statedump_block_device	This event is generated by LTTng when the tracing session starts and provides a mapping between the names and the IDs of block devices.
mark_page_dirty_IO	This event indicates that a memory page is marked as dirty for I/O.

Layer, down to the driver layer. However, in real storage systems, millions of I/O requests are processed each second, and it is almost impossible to examine them manually. Moreover, events contained in a trace file are extremely low-level and difficult to understand. It is therefore essential to use abstraction techniques to somehow summarize trace files, making

them easier to interpret.

Many abstraction techniques can be used, namely data-based abstraction, visual abstraction and metric-based abstraction.

Data-based abstraction is the most straightforward abstraction method. It consists on generating compound events by grouping low-level trace events. A hierarchy of compound events can be defined. The general behavior of the system is described by the highest level of the hierarchy, and more precisions are provided at the lowest levels. Ezzati et al. [139] proposed a stateful approach to generate synthetic events from kernel traces by using a pattern library. For example, when a process reads from a file, many low-level events are generated (system call, insertion of a request into the waiting queue, blocking the process, waking up the process when the data is available, etc). Those events can be grouped together to form a high-level event called READ_FROM_FILE, to simplify the examination of the trace. Wally et al. [140] used this approach to detect system anomalies based on aggregation patterns described on a special language that they defined. Matni et al. [141] also used the same approach to detect security related problems like SYN flood attacks.
This approach is not very efficient in the context of our work : I/O analysis requires very detailed information about lower-level mechanisms of the operating system, and using synthetic events will definitely reduce the accuracy of the analysis.

Visual abstraction is the simplification of trace data to make it fit into the screen display. This can be done by filtering insignificant events or grouping similar ones in order to reduce the complexity of the visualization. Many trace visualization tools, including TraceCompass [123] and LTTV use colored rectangles to represent the different states extracted from the trace file. Zooming and labeling techniques are also useful to show different level of details. Those techniques increase the readability of the trace, but the oversimplification may hide important details and produce misleading representations.

Metric-based abstraction consists on extracting measures from the trace file based on predefined metrics. The computed metrics provide useful information about the trace file without the need to inspect it manually. The choice of a good set of metrics is very important to get an accurate overview of the system performance. For example, measurements like disk throughput, requests latency, and request queue length are required in most disk-related performance investigations and, therefore, they should be among the collection of metrics offered by any storage analysis tool. A metric can be measured based on one or many low-level events, and it is also possible to define a hierarchy of metrics, where a high level metric can be computed from lower level metrics, which makes the computation of metrics time-consuming.
The classical way of generating statistics is to read the trace from the beginning to the end

of the selected time range, and to do the necessary calculations for this area of the trace. The problem of that methodology is that it requires rereading trace events each time the user selects a different time range. For example, calculating metrics for a specific region of a trace file necessitates reading the contained events, although they were already read when the statistics about the whole trace was generated. Keeping the trace data in memory is also not possible because trace files are generally huge and cannot fit into the main memory. Because auditing performance issues is not an easy task, users often need to go back and forth in the trace and to focus on arbitrary ranges. Spending so much time to compute the statistics at each query is not acceptable.

With those considerations in mind, we used a stateful approach where all necessary computations are done when the trace is read for the first time. A history database is incrementally built to store preprocessed metrics, which are used to quickly generate statistics for any arbitrary time range during the analysis. The generated model must be smaller in size than the original trace file so that it fits better in cache. To achieve this goal, efficient data structures and algorithms are used to effectively store and process the information. We discuss these in details in the following section.

Data Model

As explained above, our framework should support vertical and horizontal analysis. Users can examine the trace vertically, by selecting random regions of the trace, and horizontally by zooming into a specific time range to get more details. The framework must respond quickly to user queries, without rereading the trace, which could be unacceptably slow for large traces. The choice of the data structures to use is crucial to have a good performance. The data structure faces two challenges : efficiency and scalability. The response time should be reasonable, and the data structure should be able to support large trace files. Recording the state history of the system in just one pass over the trace requires a data structure that stores the data as an association between a state and a time range. For example, "The process P was blocked in the time range $[t_1, t_2]$"

Many interval management data structures like segment-tree, interval-tree and R-tree are available but most of them are memory-based, which is a major limitation for scalability. Montplaisir et al. introduced an architecture called "Modeled State System" [124] that efficiently manages the states of the different system resources in a Tree-based organization. As shown in Figure 4.4, the Modeled State System contains two main components : The Attribute Tree and the State History Tree (SHT). The Attribute Tree is used to store the hierarchy of system resources. An attribute can be accessed using a relative or absolute path.

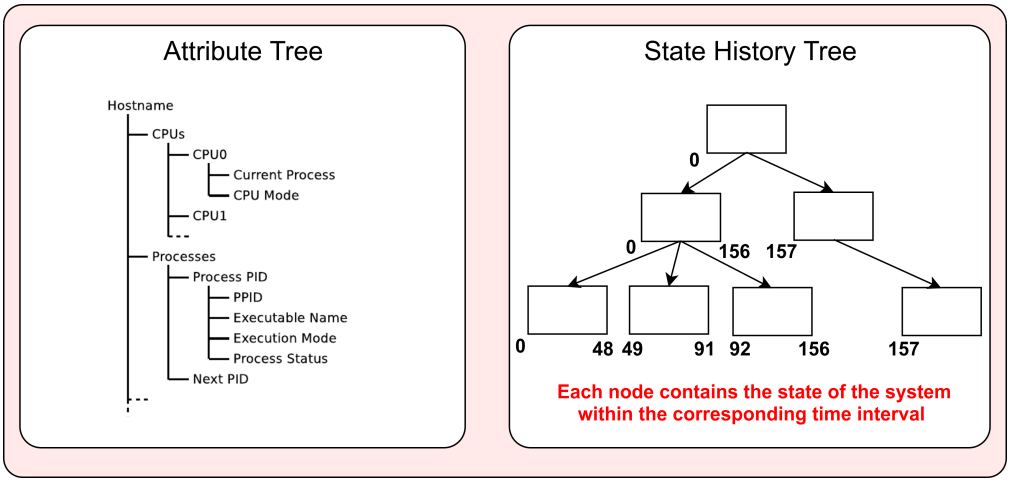


Figure 4.4 The Modeled State System Architecture

For example, the status of a thread, in a given time-stamp, can be easily extracted using the following path *System/Thread/TID/Status*. Each attribute in this tree points to an entry in the State History Tree, where the state values are actually stored. The State History Tree is a data structure optimized to keep the states of the system components over time. The tree is not balanced and is optimized to be saved on the hard disk. States are defined by a key, a value, and a time interval. Keys are used to uniquely identify system attributes, and the time interval represents the period during which the state of the attribute is equal to the given value. The data is saved to the disk in terms of nodes, the main containers of intervals. The size of nodes is a configurable parameter. Intervals are inserted sequentially into nodes. When a node is full, a new sibling node is created. A new sibling node is added to the parent if the maximum number of nodes is reached. The end time of a node is equal to the latest end time of the intervals it contains. The State History Tree can be incrementally built in one pass over the trace, making it convenient for a stateful analysis. Because it is stored on external memory, the SHT is able to support very large trace files, unlike other RAM-based data structures, which have very strict memory constraints. The State History Tree is also very efficient in terms of response time [142]. It is able to extract the state of any random attribute very quickly, compared to other tree-based data structures.

Model Structure

The model structure used in the analysis is shown in Figure 4.5. A block device has two waiting queues, an elevator queue and a dispatch queue. A request is defined by the sector number, the size and the flags. The flags indicate if the request is synchronous, asynchronous,

etc.

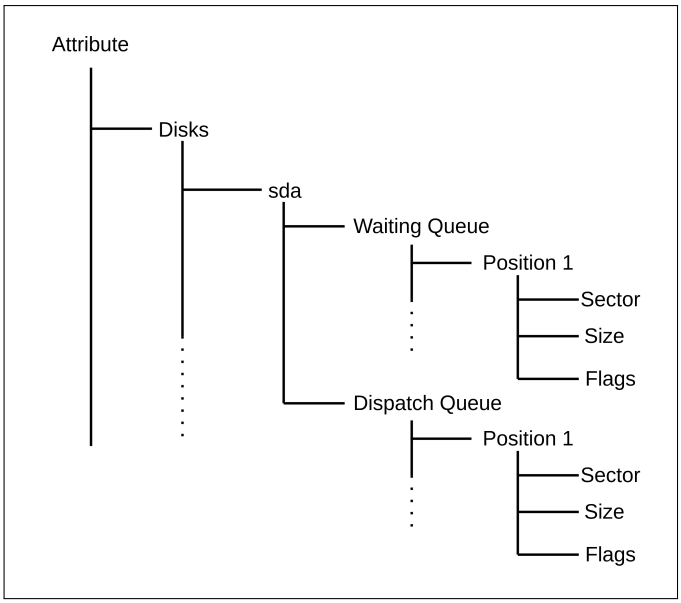


Figure 4.5 State History Tree

4.4.3 Visualization system

In a trace analysis framework, efficient data structures should be used to offer a good response time. Also, the information should be presented in a way to provide the user with a better understanding of the analyzed system. Our framework offers different views to show many aspects of the storage subsystem. The views are developed as a plugin to Tracecompass [123], an open-source trace visualization application.

The *requests view* shows the content of disk queues as a stacked timeline, where queue slots are mapped to the non-time axis and the lifetime of requests is shown as a colored rectangle. The transition of a request between different queues and merging operations are represented using an arrow that goes from the source to the destination. For example, Figure 4.6 shows an interesting pattern where a compound request (47016) is built incrementally by merging many small ones (47024, 47040, 47064, etc). The formed request is then issued to the dispatch queue for later processing.

Two views are offered to show request latencies. The *latency distribution view* (Figure 4.7) is represented as a bar chart where the x-axis shows latency ranges and the y-axis shows the number, or the percentage, of the corresponding requests.

The *latency view* (Figure 4.8) show the latencies of individual requests in a scatter chart where the x-axis represents the time and the y-axis the latency. When an outlier is detected,

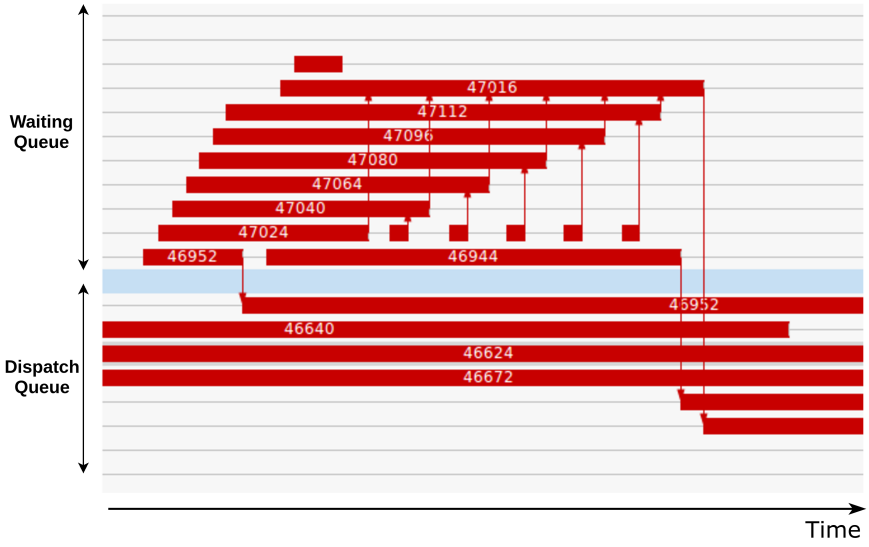


Figure 4.6 Request merging operation show in the Requests View

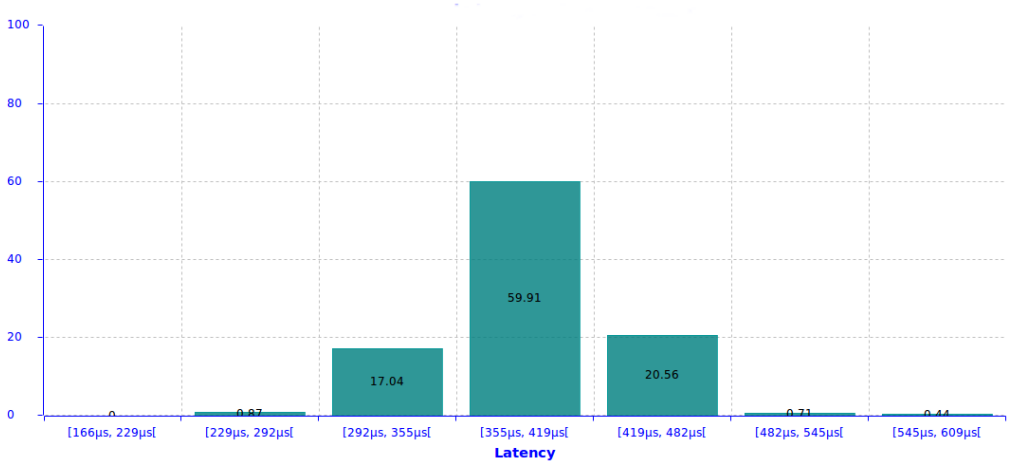


Figure 4.7 Latency distribution view

the user can get all the necessary information about the problematic request (target sector, the process that created it) by selecting its corresponding point using the mouse.

Queues lengths and disk throughput are shown as a regular time charts. All views in our framework are synchronized together. All views are updated when a user selects a new time range.

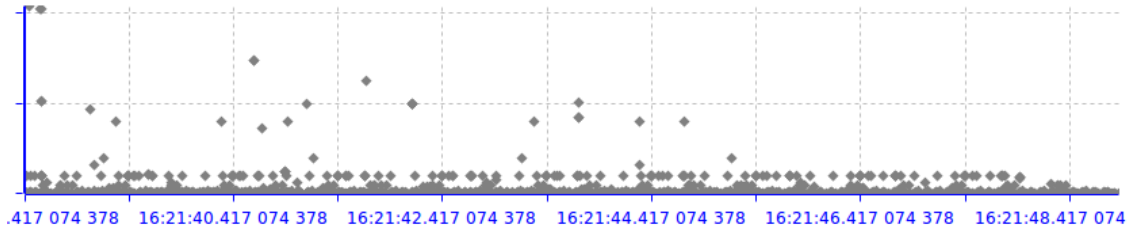


Figure 4.8 Latency Scatter Chart

4.5 Metrics Recovery

The output of a tracer is generally difficult to analyze manually. Unlike log files, trace files typically contain a huge amount of data, and the events generated are very low-level. The analysis should read the trace events and use the temporal relationship between them to synthesize useful information. In the context of a storage performance investigation, users generally rely on different performance metrics to understand what is going on in the system, and to find the origin of the issue. In the following sections, we present the request life cycle and we show how storage performance metrics can be computed from low-level trace events.

4.5.1 Request Life Cycle

An I/O request goes through different states before being processed. The different states of a request are represented as a Finite State Machine (FSM) in Figure 4.9. When a process performs a disk I/O operation, a free request slot is reserved and a request data structure is created. If all slots are occupied, the request sleeps until a slot is released. Then, the request is inserted into the waiting queue. If many adjacent requests reside in the waiting queue, the disk scheduler can decide to merge them together. Requests of different types (read and write) cannot be merged with each other. Finally, the request is issued to the driver for processing. If the disk was able to process it properly, the request data structure is released and the blocked processes are awakened. If not, the driver may decide to place the request back into the waiting queue for later processing.

4.5.2 Metrics Computation

Disk Utilization

Utilization is the average amount of time the disk was busy.

$$U = B/T$$

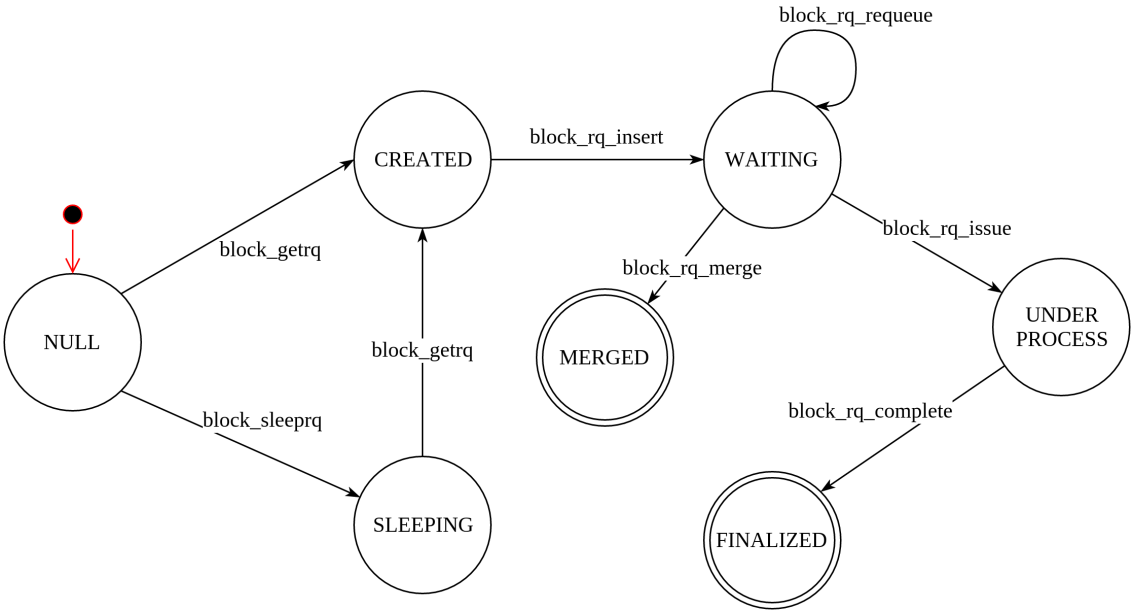


Figure 4.9 Request Life Cycle

where U is the utilization ratio, B is the amount of time the disk was busy during T, the total observation time.

A disk is considered busy if at least one I/O request is being processed in the dispatch queue.

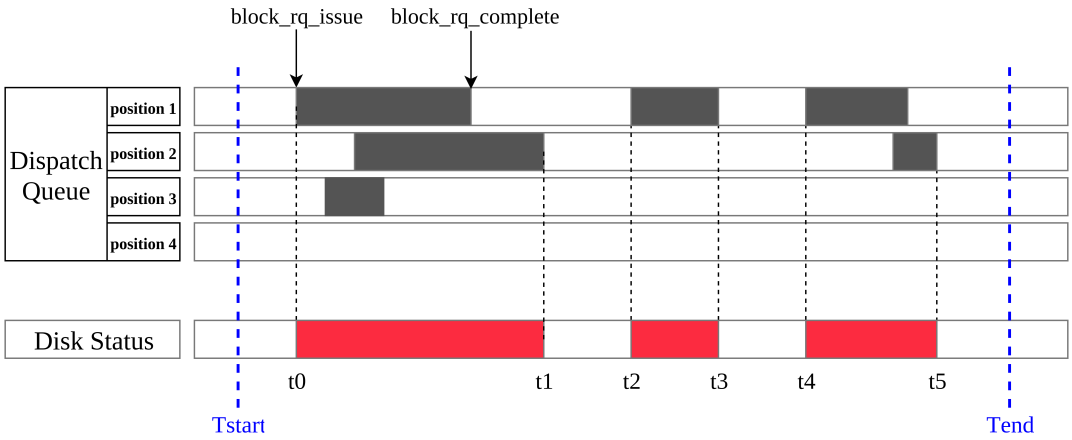


Figure 4.10 Disk Utilization

For example, disk utilization in Figure 4.10 is computed as follows :

$$U = \frac{(t1 - t0) + (t3 - t2) + (t5 - t4)}{T_{end} - T_{start}}$$

The previous formula is based on two important transitions : $Idle \rightarrow Busy$ and $Busy \rightarrow Idle$.

The first transition occurs when an I/O request is inserted into an empty waiting queue, and the second transition occurs when the last I/O request in the queue is processed. If we consider that these transitions occur at t_s and t_e respectively, the general formula becomes :

$$U = \frac{\sum_i (t_{s_i} - t_{e_i})}{T_{end} - T_{start}}$$

Latency

The latency is the time that an I/O request takes to be processed by the disk drive. As illustrated in Figure 4.11, the latency can be divided into three parts : preparation time, waiting time, and service time [143]. Preparation time is the time needed to create and to initialize the request data structure. The waiting phase starts when the request is inserted into the waiting queue and ends when it is dispatched for processing. Service time is the time that the disk drive needs to finish processing the request. It is important to mention that service time depends on the request size and on the disk speed, whereas preparation and waiting times are mostly affected by both the filesystem and the disk scheduler.

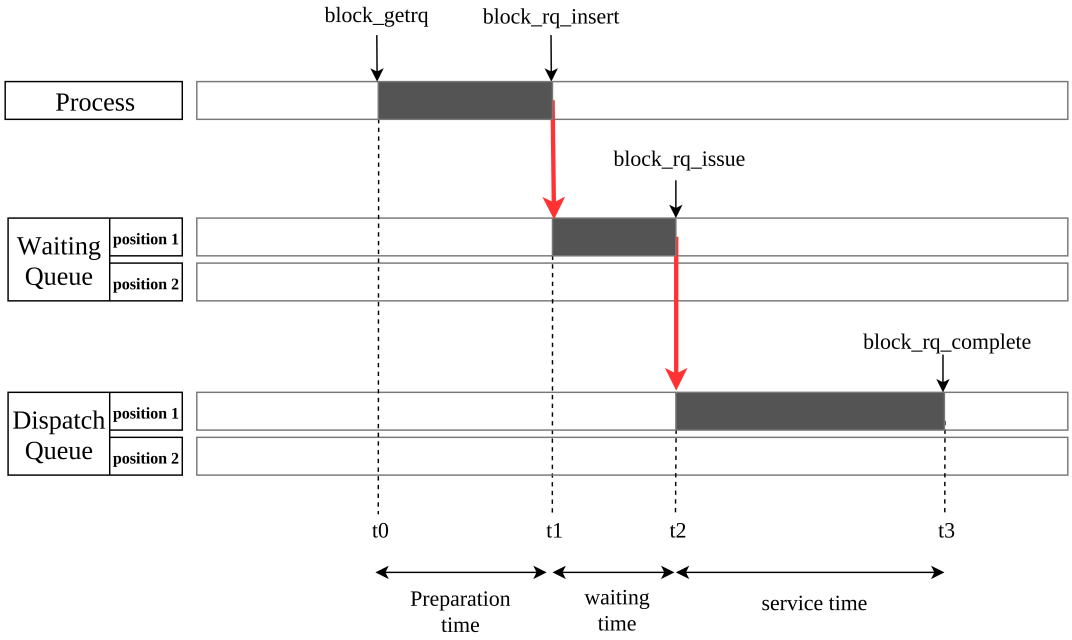


Figure 4.11 Latency

The disk schedule may merge two requests together if they are adjascent. *elv_merge_requests* is the function used by the Linux Kernel to do the merging. It takes the origin and the destination requests as arguments. By instrumenting this function, we are able to track the completion time of the requests as well as the contribution of the different processes.

Another interesting case is to be considered. I/O operations are not always processed in the context of the process who issues them. Sometimes, a request is processed asynchronously by the *pdflush* daemon. In this case, it is important to find a way to know the process responsible for the operation. We manage that by tracing the function *mark_page_dirty_IO* which is called when a process marks a memory page as dirty for later processing. When the *pdflush* starts to write back the data to the disk, we know exactly the processes responsible for each asynchronous I/O operation.

Seek Time

In magnetic disks, seek time is the time that the mechanical disk head takes to reach the target sector. In the case of random workloads, seek time is higher than transfer time, which means that the disk drive takes more time to locate the data than to transfer it. Consequently, a sequential workload is much more efficient than a random one, since the disk head doesn't have to travel frequently from one position to another. Seek time is proportional to the travel distance. Computing the exact travel distance requires precise information about the internal architecture of the disk such as the number of platters and the number of sectors per track. To make the analysis generic and architecture-independent we used the following approximation : If an I/O operation on a logical block address LBA_s is followed by another operation on LBA_e

$$Sectors\_travelled = |LBA_e - LBA_s|$$

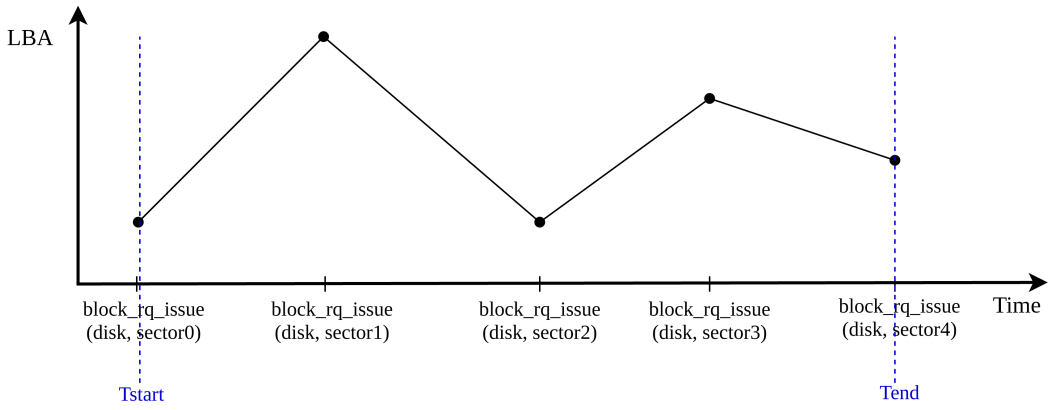


Figure 4.12 Sectors traveled

In figure 4.12, sectors travelled between T_{start} and T_{end} can be measured as follows :

$$Sectors\_travelled = |sector_1 - sector_0| + ... + |sector_4 - sector_3|$$

In general

$$Sectors\_travelled = \sum_i |sector_{s_i} - sector_{e_i}|$$

Queue length

The queue length computation is straightforward. The counter is incremented when a request is inserted into the queue and decremented when it is removed from it. The queue length is also decremented when two requests are merged together, as illustrated in Figure 4.13.

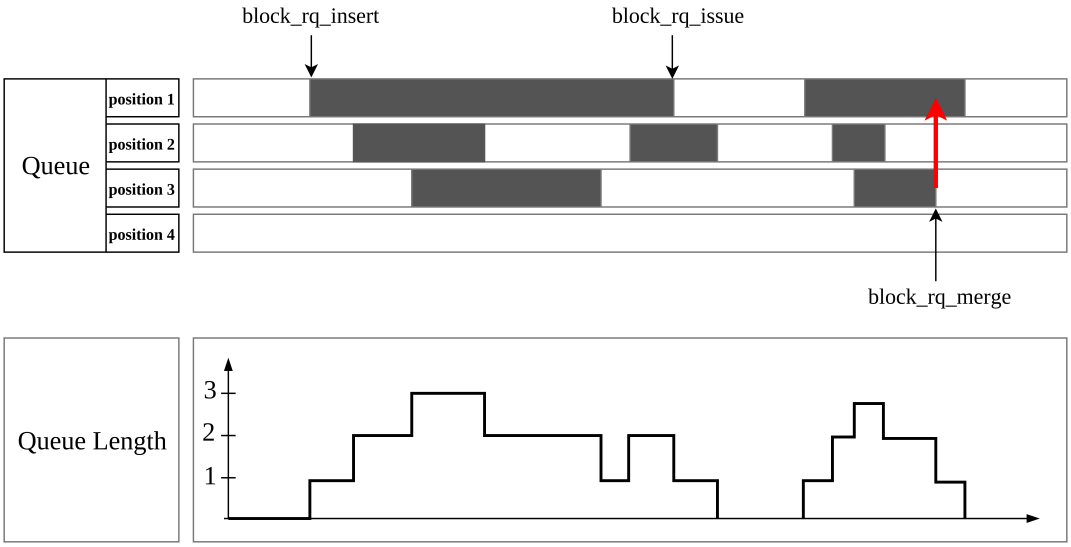


Figure 4.13 Queue Length

The size of requests is also a metric to be considered. A large number of small-sized scattered requests is a symptom of a random I/O workload that should be avoided, if possible.

4.6 Use cases

The proposed analysis framework provides different metrics and views to facilitate storage debugging tasks. In this paragraph, we show how our framework can be used to solve difficult real world problems.

4.6.1 Investigating a high latency

A web server for which users are complaining about the response time was analyzed with our storage analysis framework. The server, which is running Apache 2.4 and Linux Kernel

4.4, suffers from a very big variability in response time. Figure 4.14 shows that a web page sometimes takes up to 200 ms to be processed, which is unexpectedly high compared to the 20 ms normal response time. To reproduce the problem, a HTTP crawler that simulates visitors activity was executed. LTTng is used to gather execution traces when the problem occurs.

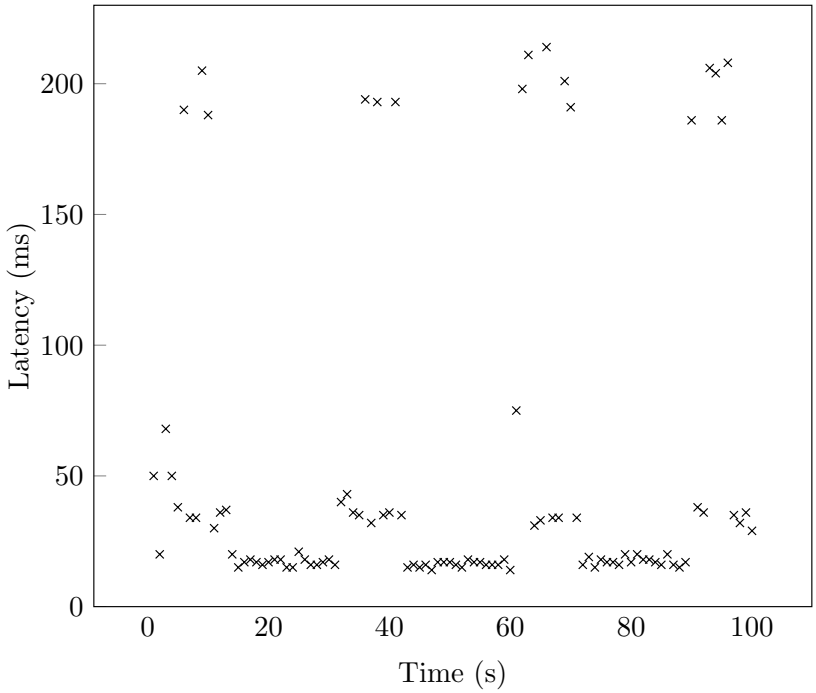


Figure 4.14 The web server response time

Based on trace events, our framework provides detailed information about request latency. As explained above, latency can be divided into preparation time, waiting time and service time. The analysis shows that a problematic request spends a long time in the waiting queue, whereas normal requests are immediately issued for processing (Figure 4.15). Accordingly, we deduce that the problem is caused by a wrong scheduling decision.

The request view of the framework (Figure 4.16) shows the content of the disk queues at any given time. This view shows that while an Apache read request (shown in red) resides in the scheduler waiting queue, many other requests (shown in light red) are getting processed immediately. Those requests are high priority requests inserted by *backup.sh*, a process that runs periodically to backup server files.

In conclusion, the problem can be stated as follows : the web server requests are being interrupted by higher priority requests issued by a backup service. This behavior is problematic because the backup service should not have a dramatic impact on the server. By switching the

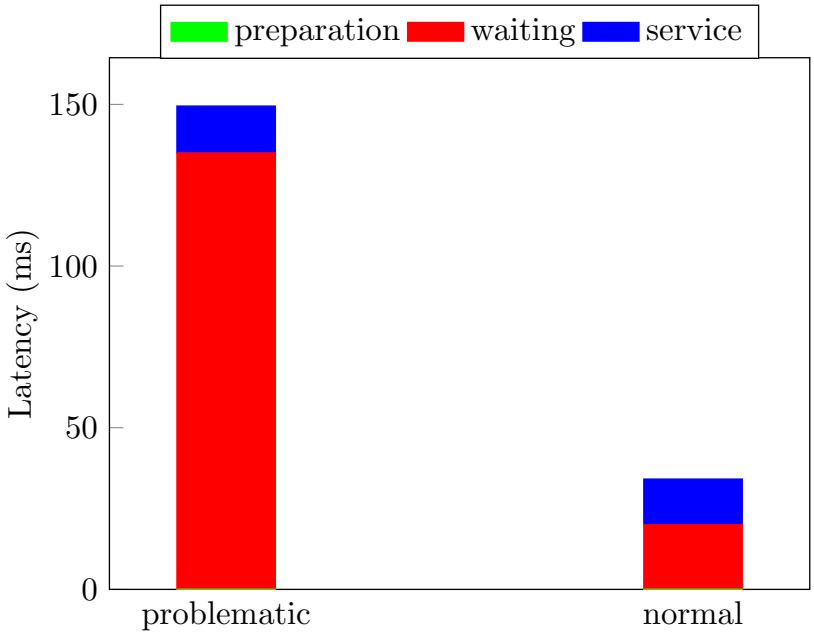


Figure 4.15 Latency details of a problematic web request

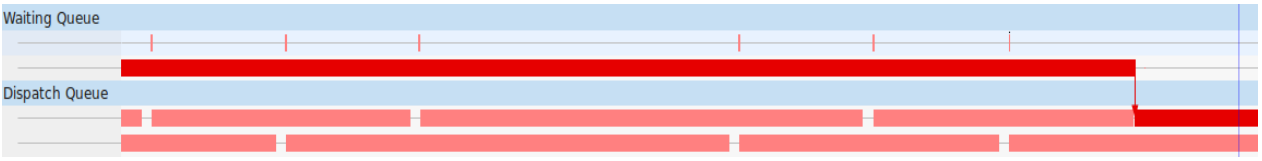


Figure 4.16 The waiting queue content during a problematic request

priorities of Apache and *backup.sh*, the latency of a web request during the backup process was reduced from 200 ms to 50 ms, as shows Figure 4.17.

4.6.2 Investigating a data loss

In this section, we investigate the cause of data loss caused by a power outage shortly after writing data

The script *backup.sh* presented above is a *cron* job that runs periodically to copy the web server files to a safe directory using the Linux *dd* command. After a sudden power failure, many files that are supposed to be backed up are lost. In this section, we show how our framework was used to understand the problem and to resolve it.

The system calls view shows that the *dd* command copies a file by calling read and write system calls successively (Figure 4.18). In each iteration, the process reads data from the

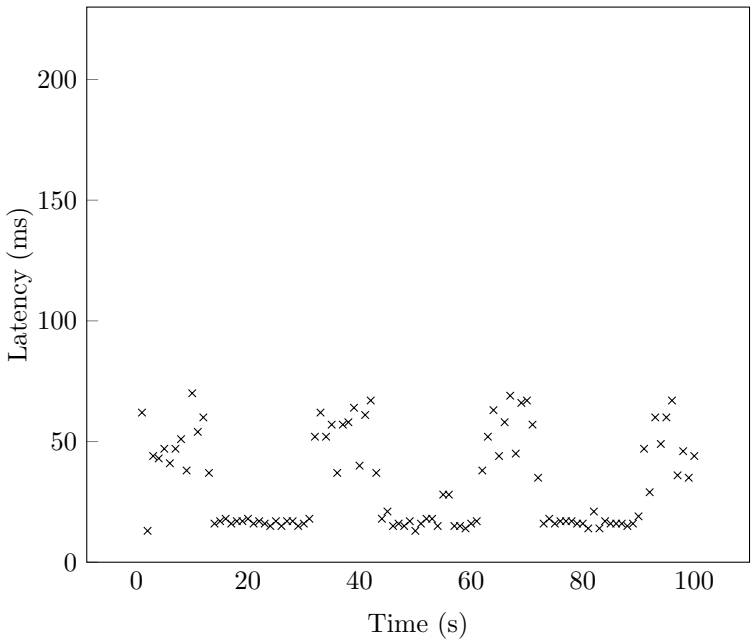


Figure 4.17 The web server response time after switching the priorities

original file and writes it to the new file. read system calls are processed synchronously : the orange-colored area in the view shows that the *dd* process gets blocked during the system call to wait for the required data. On the other hand, write system calls are processed asynchronously : memory pages are marked dirty in the page cache and the system call returns immediately. A kernel thread, called *pdflush*, is run periodically by the operating system to insure that writes are not delayed for more than a certain threshold defined in */proc/sys/vm/dirty_expire_centisecs*. If a power failure occurs before the data is effectively written to the physical drive, the data available in the page cache is lost.

Figure 4.18 System calls executed during a file copy operation

The disk throughput view of the framework confirms that no writing activity is observed during the execution of the *dd* command(Figure 4.19).

In critical applications, like backup processes, it is important to make sure that data is written to the disk as soon as possible. The O_SYNC flag, offered by the POSIX library, should be used to insure that the data, and all required meta-data, are written to the physical disk synchronously by issuing reading and writing requests alternately (Figure 4.20). An

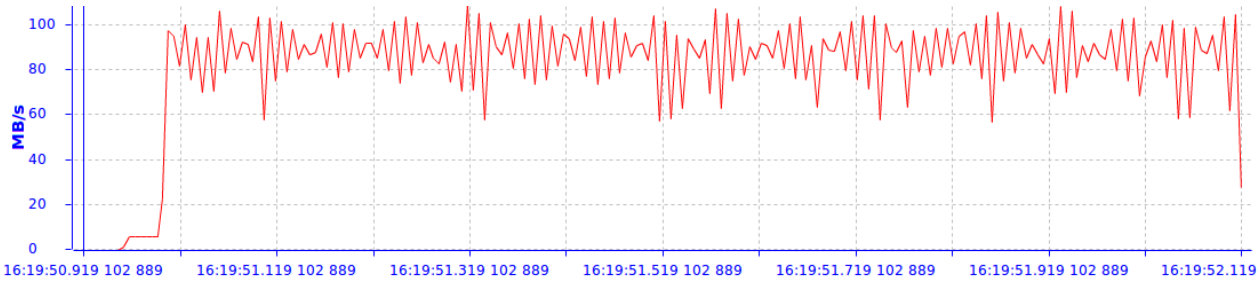


Figure 4.19 Disk throughput during a file copy operation

alternative solution is to use the *fsync* system call to flush the page cache to the disk drive (figure 4.21). *Fsync* is usually considered as the preferred alternative because it involves less disk seeks and offers a better throughput.

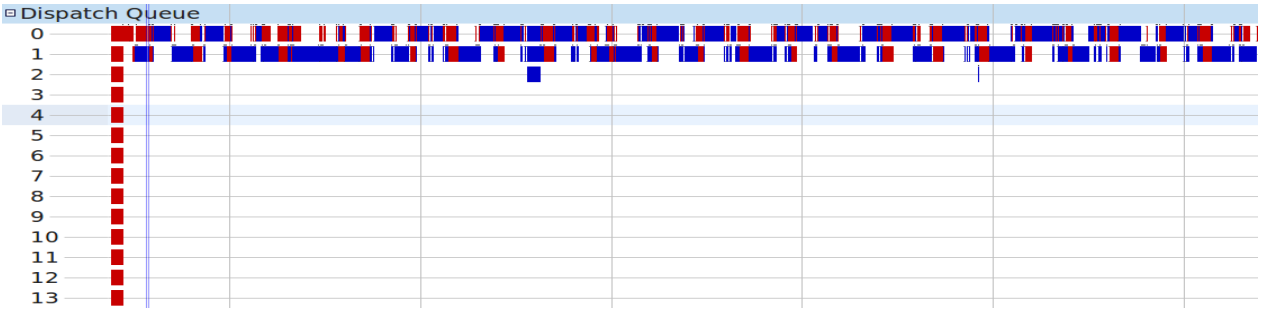


Figure 4.20 Synchronous file copy

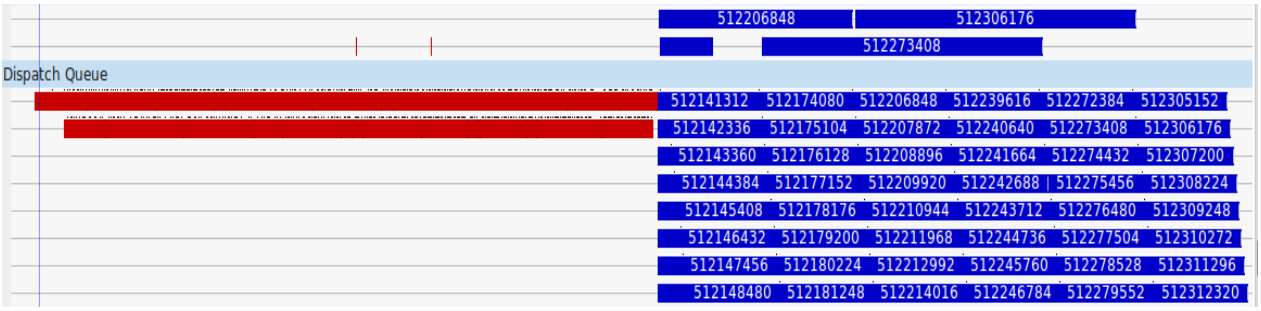


Figure 4.21 File copy + *fsync*

Surprisingly, even after making sure that writing requests are issued to the disk controller, whether by the *pdflush* thread or with an explicit call to *fsync*, data corruption was detected.

In fact, almost all modern disk drives offer a fast volatile memory that behaves as a write-back cache. I/O request buffers are firstly written to the cache before being asynchronously transferred to the non-volatile memory. The low response time offered by the write-back cache explains the peak in disk throughput at the beginning of the writing operation (figure 4.22)

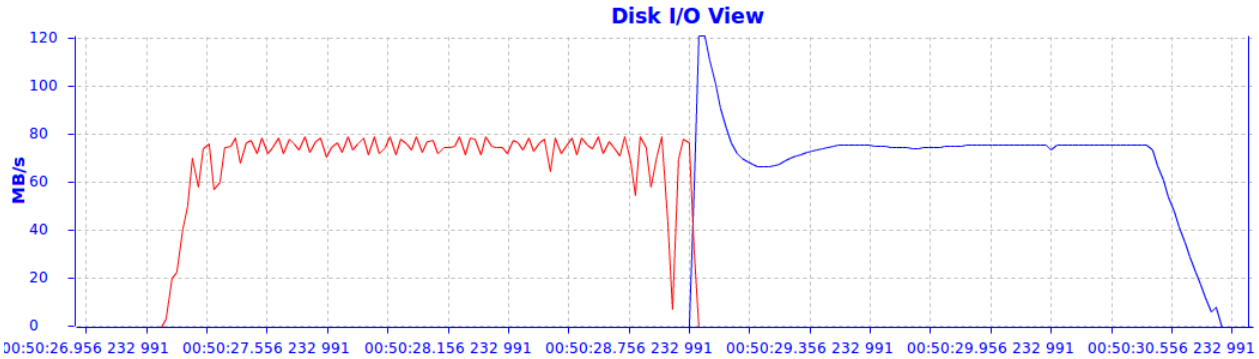


Figure 4.22 Disk Writeback Buffer

Write-back caches come with a major drawback. An I/O completion signal is sent to the operating system when the data is written to the cache, not to the physical disk. As a result, a data loss can occur in the case of a sudden power failure. To avoid this situation, *fsync* must issue a flush request to the disk drive to flush the write-back cache just after flushing the page cache. The request view (Figure 4.20) revealed a problem : no flush request is issued to the disk at the end of the *fsync* system call. After some investigations, we figured out that the filesystem is configured with a wrong mount option (*nobarrier*) which should only be used if the disk does not contain a write-back cache. Mounting the ext4 file system with the *barrier* mount option resolved the problem, as shown in Figure 4.23.

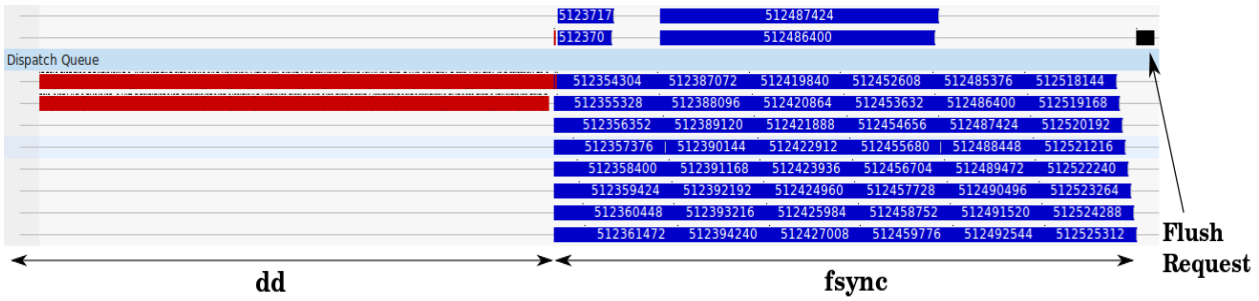


Figure 4.23 Flush request waiting queues

When a flush request is processed, the dispatch queue is blocked and all the new incoming requests are accumulated in the waiting queue (Figure 4.24)

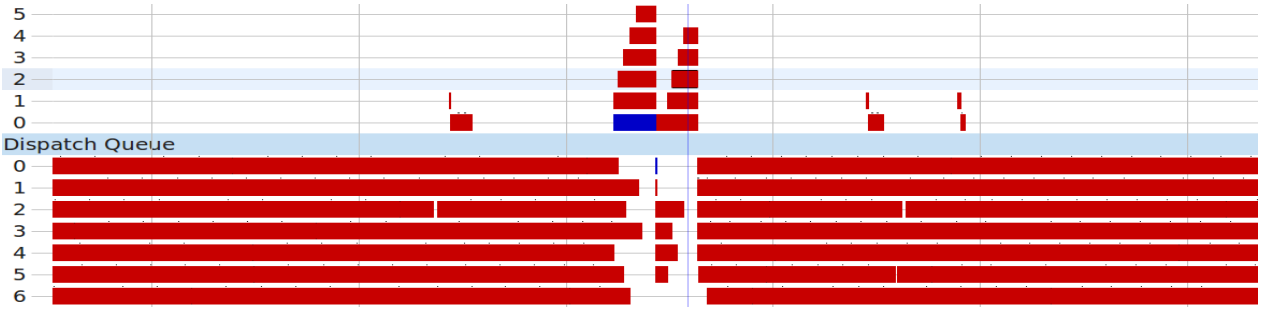


Figure 4.24 Flush requests

4.7 Performance

In this section, we evaluate the impact of our framework on the system performance. The evaluation is done by calculating the additional overhead of tracing on an I/O intensive workload. In addition, we study the time and the disk space required for the analysis.

4.7.1 Test Setup

To prove the low overhead of tracing under any circumstance, we performed tests on very diverse workloads and under different configurations. The workloads are generated using SysBench, a benchmark suite available on Linux [144]. The tests are performed on an Intel i7-4790 CPU @ 3.60GHz with 32 GB of main memory, running Linux Kernel version 4.4. The hard disk used for benchmarking is an Intel SSD 530 Series 240 GB. The traces are gathered using LTTng 2.8.

1. Workloads :
   - *Sequential Read :* Files are read sequentially from the beginning to the end.
   - *Sequential Write :* Files are written sequentially from the beginning to the end.
   - *Random Read :* Data is read randomly from files.
   - *Random Write :* Data is written randomly to files.
   - *Random Read/Write :* Read and Write operations are performed randomly from files. Read/Write ratio is 1.5.
2. Benchmark configuration :

— *Base :* The tests are run with tracing deactivated. This configuration is used as the reference for overhead computations.
— *External :* The trace file is written to an external disk drive to eliminate the interference between the workload execution and the tracer output.
— *Internal :* The trace file is written to the same disk on which the workload is executed.

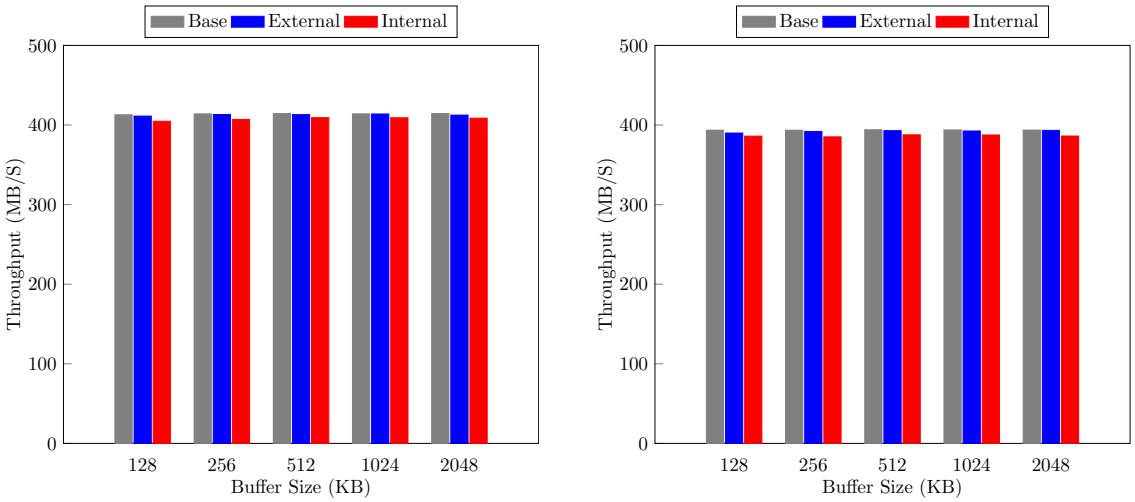
Because the granularity of I/O operations depends on the buffer size, we executed each test with four different buffer sizes : 256 KB, 512 KB, 1 MB, and 2 MB
The workload size involved in each test is 20 GB. To avoid the effect of caching, we created a RAM filesystem (RamFS) of 31 GB, so that only 1 GB of free memory is left to the system (5% of the workload size). Each execution is run ten times and the average and standard deviations are provided.

4.7.2 Tracing Overhead

Benchmarking results are provided in Table 4.2, Figure 4.25, and Figure 4.26.

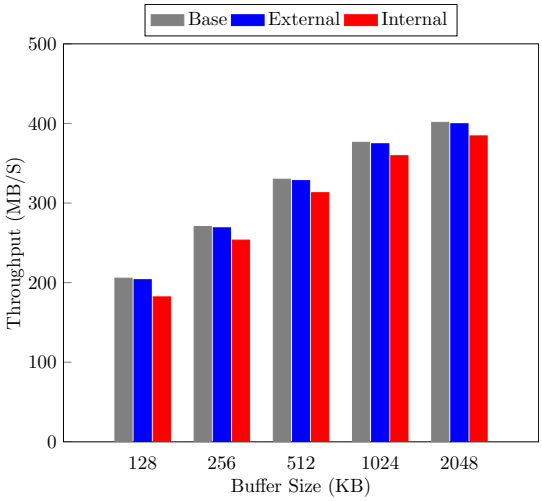
As expected, the throughput of the disk is much better when the workload is sequential. In fact, a random workload involves a higher number of seek operations than does a sequential workload, considerably affecting the performance of the disk. Reducing the buffer size increases the randomness of the workload, which explains the fact that the worst throughput is observed with a buffer size equal to 128 KB. The buffer size does not have an impact on sequential workloads thanks to the readahead algorithm : the disk drive starts to automatically prefetch the data from the disk when a sequential workload is detected.

As for the tracing overhead, Figure 4.26 shows that the overhead of external tracing doesn't exceed 1.1% for all workloads. On the other hand, internal tracing introduces a higher overhead of about 2% for sequential workloads and up to 11.4% for random workloads. It is clear that the degree of randomness also affects the tracing overhead : the overhead increases when the buffer size decreases. With small buffer sizes, the number of created I/O requests is enormous and, thus, the throughput of the tracer is very high. The overhead of internal tracing can be unacceptable if the applications are performing very random workloads. This case is not very common since these kinds of applications usually use a userspace cache to reduce the number of I/O operations. Yet, we recommend using an external storage for saving the trace, especially in production systems.

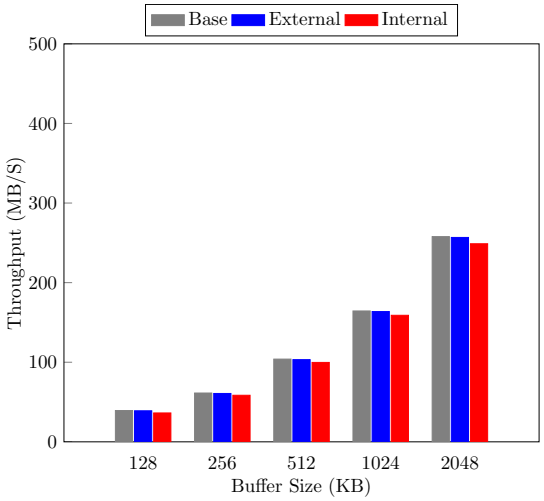


(a) Sequential Read

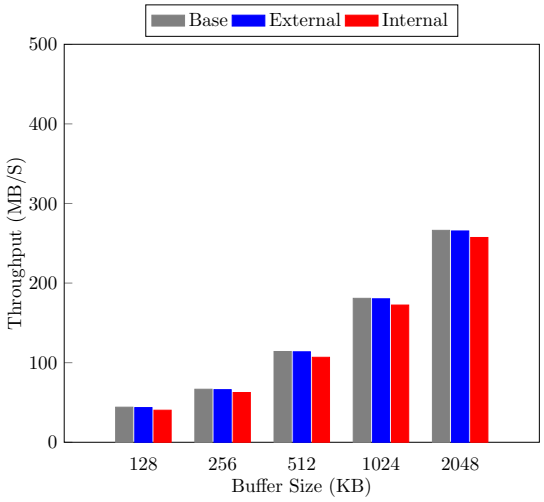
(b) Sequential Write



(c) Random Read

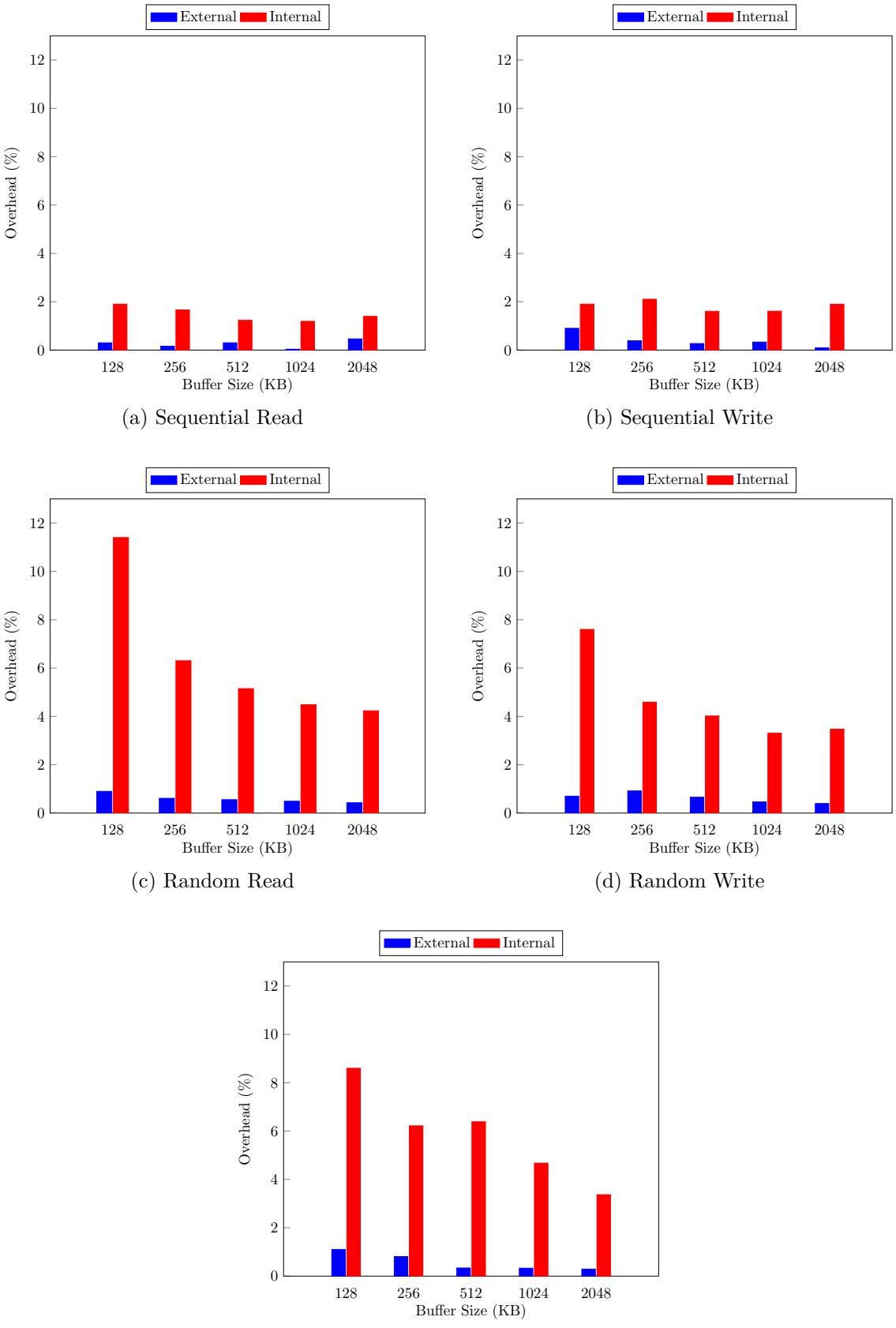


(d) Random Write



(e) Random Read/Write

Figure 4.25 Throughput



(a) Sequential Read

(b) Sequential Write

(c) Random Read

(d) Random Write

(e) Random Read/Write

Figure 4.26 Overhead

Table 4.2 Tracing overhead benchmark

Workload	**Block Size (KB)**	**Base ave (MB/s)**	**Base Std.dev (MB/s)**	**External ave (MB/s)**	**External Std.dev (MB/s)**	**Internal ave (MB/s)**	**Internal Std.dev (MB/s)**	**External Overhead (%)**	**Internal Overhead (%)**
Sequential Read	128	412.8	0.5	411.2	0.8	404.7	0.6	0.3	1.9
	256	413.9	0.7	413.2	0.9	407.0	5.6	0.1	1.6
	512	414.3	0.9	413.0	1.1	409.2	0.8	0.3	1.2
	1024	413.9	0.8	413.8	0.7	409.0	0.7	0.0	1.1
	2048	414.3	0.9	412.4	1.2	408.5	3.9	0.4	1.3
Sequential Write	128	393.5	1.1	389.9	1.2	386.0	0.3	0.9	1.9
	256	393.4	0.7	391.9	0.7	385.1	0.6	0.3	2.1
	512	394.0	0.5	392.9	1.0	387.6	0.8	0.2	1.6
	1024	393.7	0.7	392.4	1.8	387.4	0.5	0.3	1.6
	2048	393.6	0.4	393.2	0.5	386.1	2.6	0.1	1.8
Random Read	128	205.6	0.4	203.7	0.3	182.1	0.5	0.9	11.4
	256	270.5	0.5	268.8	0.3	253.4	1.2	0.6	6.3
	512	329.9	0.1	328.1	0.2	312.9	0.3	0.5	5.1
	1024	376.3	0.4	374.4	0.4	359.4	0.3	0.4	4.4
	2048	401.3	0.1	399.6	0.3	384.3	0.4	0.4	4.2
Random Write	128	39.3	0.8	39.0	0.7	36.3	0.4	0.7	7.6
	256	61.3	0.4	60.7	0.1	58.5	0.2	0.9	4.5
	512	103.9	0.1	103.2	0.2	99.7	0.6	0.6	4.0
	1024	164.4	0.2	163.6	0.4	158.9	0.6	0.4	3.3
	2048	257.8	1.4	256.7	0.6	248.8	1.09	0.3	3.4
Random Read/Write	128	44.1	1.1	43.6	0.8	40.3	0.6	1.1	8.6
	256	66.6	0.4	66.12	0.7	62.5	9.4	0.8	6.2
	512	114.0	0.5	113.6	0.6	106.7	0.6	0.3	6.3
	1024	180.8	0.7	180.2	1.4	172.4	0.8	0.3	4.6
	2048	266.2	2.1	265.4	1.4	257.2	0.9	0.2	3.3

Table 4.3 Analysis Cost

Workload	Block Size (KB)	Execution Time (s)	Trace Size (MB)	Trace Reading Time (s)
Sequential Read	128	49.6	124.0	22.8
	256	49.5	122.0	21.5
	512	49.4	112.0	19.2
	1024	49.5	99.0	18.1
	2048	49.4	106.0	18.9
Sequential Write	128	52.0	109.0	19.1
	256	52.1	113.0	21.9
	512	51.9	96.0	16.6
	1024	52.0	98.0	16.8
	2048	52.0	91.0	16.1
Random Read	128	99.6	185.0	34.2
	256	75.7	151.0	26.5
	512	62.0	117.0	21.8
	1024	54.4	94.0	17.6
	2048	51.0	83.0	14.7
Random Write	128	521.1	164.0	30.6
	256	334.0	139.0	26.5
	512	197.1	101.0	18.8
	1024	124.5	86.0	15.1
	2048	79.4	75.0	13.6
Random Read/Write	128	464.3	172.0	31.5
	256	307.5	143.0	26.7
	512	179.6	108.0	19.3
	1024	113.2	91.0	17.1
	2048	76.9	79.0	14.7

4.7.3 Analysis Cost

In this section, we evaluate the performance of our analysis tool for the different workloads presented above. Two dependent variable are considered : trace size and trace reading analysis time.

The results presented in Table 4.3 show that the trace size and the reading time depend on the randomness of the workload. In the benchmark performed, the trace size reaches 124 MB for sequential workloads and 185 MB for random workloads. The reading time is relative to the trace size. It reaches 22.8 seconds for sequential workloads and 34.2 for random workloads. Pure random workload traces are big and require more time to be analyzed because the number of recorded events is very high, but it is still acceptable since the analysis is done offline.

4.8 Conclusion

Storage debugging is a complex task that needs a deep understanding of the internals of the operating system and the interaction between its different components. In this paper, we proposed a storage analysis framework based on kernel traces that helps in debugging storage related issues. Because the output of tracing is usually enormous and unintelligible by a trouble-shooter, the main effort was on computing relevant performance metrics and providing a comprehensive user-friendly visualization system. Efficient algorithms and data structures are provided to ensure a fast query time. The approach is based on computing and storing the state of the system in a tree-like data structure, where each node corresponds to a system resource, in just one pass over the trace. Performance metrics can be computed by querying the state system and performing basic interpolation techniques.

The results denote that the framework does not add a significant overhead to the system, which makes it suitable for production systems. Use cases are also provided to show the usefulness of the proposed framework in real situations.

The framework presented in this paper is mostly integrated into the official versions of Lttng and Tracecompass. Some features are still under testing and are actually available in the GitHub reporsitory of the author [1].

A possible future work is to make the framework more generic by supporting other tracing systems like Solaris DTrace and Windows ETW. It is also important to include the support of live tracing to permit online monitoring. Another interesting improvement is to provide an

1. https ://github.com/houssemmh/

API allowing the extension of the framework by defining new patterns of fault detection. For example, a user can ask the framework to generate an alert when the latency or the queue length exceed a certain value.

Acknowledgments

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson and EfficiOS.

CHAPITRE 5 ARTICLE 2 : PERFORMANCE ANALYSIS OF DISTRIBUTED STORAGE CLUSTERS BASED ON KERNEL AND USERSPACE TRACES

Authors

Houssem Daoud and Michel Dagenais
Department of Computer and Software Engineering, Ecole Polytechnique de Montreal, Montreal, H3T 1J4, Canada

E-mail : {houssem.daoud, michel.dagenais}@polymtl.ca

Submitted to Software : Practice and Experience

5.1 Abstract

Distributed storage systems are commonly used in modern computing. They are highly scalable and offer data replication and fault tolerance. The complexity of those systems makes them difficult to debug using traditional tools. The existing tools are able to evaluate the overall performance of such systems but they do not provide enough information to find the root cause of performance issues. In this paper, we propose a tracing-based performance analysis framework for storage clusters. We use a tracing strategy that reduces the tracing overhead in production systems. The traces collected from the different storage nodes are correlated and used to generate a data model that represents the cluster. Userspace tracing is used to gather data from the storage daemons, while Kernel tracing is used to provide detailed information about operating system internals such as disk queues, network queues and process scheduling. Efficient data structures are used to store the model and to generate metrics and graphical views. Our tool is used in different real world scenarios and is able to investigate interesting performance problems including I/O latencies, data replication and storage nodes failures.

5.2 Introduction

The advent of Cloud computing and Big Data increased the motivation for developing storage solutions that offer performance, reliability and fault tolerance. Traditional storage solutions based on huge data centers are not convenient, because they offer very limited scalability in

terms of available space and the number of clients that can access the data concurrently.

Distributed storage systems, such as Ceph [57] and Lustre [54] , were proposed to solve scalability issues by storing the data in a cluster of networked storage nodes. The distributed model offers many advantages. Firstly, the number of storage nodes can scale dynamically, depending on storage needs. It is possible to add or remove a storage node dynamically, without causing a service interruption. Secondly, files can be replicated in many storage nodes, located in different physical locations, which insures data safety in case of a power failure or natural disaster.

Distributed storage systems use advanced techniques to satisfy the needs of modern applications, which makes the performance analysis of such systems a challenging task. The complexity of those systems affects the system administrators by impeding their ability to understand and debug their clusters. Performance bottlenecks can be caused by many things such as the network, the disk or deployment misconfigurations.

Benchmarking tools like *CBT* [145] and *Rados Bench* [146] evaluate the performance of the cluster by running different workloads and computing the execution time. Those tools are limited because they use synthetic workloads and are unable to analyze the performance of the cluster in real world scenarios. On the other hand, monitoring tools [147] [148] provide high-level information about the cluster health. They periodically collect performance metrics provided internally by the the cluster and show them in different graphical views. Those tools suffer from two major limitations. Firstly, the frequency of data collection is fixed beforehand, and as a result some important events may be missed. Secondly, the collected data is limited to userspace and does not provide any low-level information about the system on which the storage daemons are running. Having low-level details about the operating system and the hardware involved is very important to detect complex performance issues. For example, an I/O request may be slow because the waiting queue of the disk is full, and this information is not accessible from the userspace level.

In this paper, we propose a performance analysis framework that provides a full visibility of the system, by collecting Kernel and userspace traces. The traces contain information about the cluster daemons as well as low-level information gathered from from the operating system components, including the CPU scheduler, the I/O scheduler, the network layer, etc. Having the overall state of the system allows detecting the root cause of complex problems, that other tools are unable to detect. The amount of information collected in traces may be huge. Consequently, we provide an efficient architecture to minimize as much as possible the generated data. Our tools first trace the execution at a high level, to monitor and find the problematic latencies. Then, it enables detailed tracing when something wrong or suspicious

happens. The detailed tracing data contains a significant amount of runtime data, to be investigated in depth later on, about the time interval during which the problem was detected.

The proposed solution was implemented using LTTng [3], a low-overhead tracer available on Linux. We have chosen Ceph as the reference distributed storage system in this paper, because it is widely used in industry, open-source, and implements most of the technologies used in modern distributed storage systems.

The contributions of this paper are the following :

1. An efficient architecture for a lightweight execution tracing, while detailed tracing, at the different layers, is generated for problematic areas.
2. A model to organize the data collected from the different layers, to be used during the post-portem analysis.
3. A set of analysis algorithms and visualization views to understand and debug the runtime behaviour of distributed storage requests.

The rest of the paper is organized as follows : in section 2, we present the existing distributed storage systems and we provide a literature review about the performance analysis of distributed storage systems. In sections 3 and 4, we describe the architecture of the proposed framework. Then, we present different use cases in section 5 and we evaluate the overhead of tracing in section 6. Then, the paper concludes with a discussion about the challenges addressed, and possible future work.

5.3 Litterature review

In this section, we present the most popular distributed storage systems and we discuss the design challenges behind those systems. Then, we present Ceph and review the different studies related to Ceph performance analysis.

5.3.1 Design challenges of distributed storage systems

Traditionally, storage devices were directly connected to the motherboard of the machine on which the programs are running. This architecture is very rigid and cannot meet the needs of modern computing. One of the first widely used remote access protocol for storage devices was the Network File System (NFS) [12]. NFS is a client-server architecture where clients are able to mount a filesystem over the network. This solution is not scalable and is typically used for file sharing among a small number of clients and storage nodes.

Many distributed storage solutions like OceanStore [47] and Farsite [48] are proposed for very large scale systems. The idea behind those architectures is being able to create a huge

storage environment, composed of untrusted storage nodes formed by clients. They use Byzantine agreement protocols to protect data against malicious clients, which introduce a large overhead and make the system inappropriate for high performance computing.

With the increased usage of parallel programming, the focus has shifted from the capacity of storage devices to the performance they offer, in terms of bandwidth and response time. A distributed storage system should be able to scale in terms of storage capacity, without paying a big performance penalty for each additional storage node. To increase the storage bandwidth, PVFS [50] stripes file data across multiple storage nodes, so that the different parts of a file can be accessed simultaneously by different clients. Metadata information is kept in a single manager that keeps the mapping between files and their placement in the cluster. Vesta [51] uses a similar striping technique, but in addition offers to users to explicitly create partition maps that match the parallelism in their application. The bottleneck of PVFS and Vesta is mainly metadata lookup. The clients must contact the metadata server before being able to store or retrieve data from storage nodes. The use of caching can reduce the frequency of lookups but still, having a centralized metadata server in the critical path of I/O operations is not a viable solution in a large scale distributed storage system.

GPFS [52] partially decouples data and metadata accesses by introducing the concept of *metanode*. One of the nodes accessing the file is dynamically selected as a metanode and it is responsible for updating the metadata. shared write locks are used, instead of exclusive locks, to avoid contention on every write operation. Arumugam and al. [149] studied the metadata bottleneck in pNFS [150] and proposed to distribute the metadata across a cluster of nodes.

The object-based storage paradigm was proposed to solve the scalability problem. Instead of representing the data as block ranges, and relying on the filesystem to describe it, the data is represented in terms of objects. An object contains the data itself and its associated metadata, which allows the system to operate at the petabyte scale and offers better mechanisms for failure recovery and data replication. zFS [53] and Lustre [54] are object-based storage systems designed to support several thousands of object storage nodes (OSDs). OSDs are constructed from commodity hardware, running an operating system and containing a CPU, a network interface and disk drives. zFS and Lustre suffer from two main problems. Firstly, they use a partially distributed metadata management which becomes a bottleneck at a very large scale. Secondly, they do not use the intelligence provided by the OSDs to help maintaining the storage system.

Sorrento [56] fixed the problem of metadata management by using the consistent hashing algorithm. Instead of using explicit allocation tables, like other distributed systems, the algorithm is executed to determine data placement, removing the metadata lookup for the

critical path of I/O operations. Sorrento supports storage node joins and departures, but it does not offer the flexibility to adapt the algorithm to the physical architecture of the cluster.

5.3.2 Ceph

Ceph [57] is a distributed storage system that was proposed to solve the different challenges detailed in the previous section. It gained a lot of popularity in recent years because it offers the different services required by modern applications. It provides a simple object storage, for large-scale data storage and backup services, a block storage abstraction RBD (Rados Block Device) for virtual machines, and CephFS, a regular POSIX-like file system, for other usages. Those services are built on top of a RADOS cluster (Reliable Autonomic Distributed Object Store) [59] and can be provided simultaneously without changing the setup.

Data is striped across storage units represented by Object Storage Devices (OSD). An OSD daemon can be executed on any commodity machine containing a network interface and a block storage device, and takes the responsibility of internal block request management and scheduling. The placement of objects is defined by CRUSH (Controlled Replication Under Scalable Hashing), a hash-based algorithm that pseudo-randomly distributes data across storage devices. Using this algorithm removes the need to perform a name lookup in a central directory before every I/O operation, and allows clients to perform high-performance parallel data accesses.

System administrators are able to define the physical distribution of their data using a CRUSH map. The map is kept by a small cluster of monitors and it describes the storage nodes hierarchy as well as the data placement and replication policies. Storage nodes can be added or removed dynamically, without causing a large data migration for re-balancing. One of the biggest strengths of Ceph is leveraging the intelligence provided by OSDs to help maintaining the health of the cluster, and performing necessary actions in case of a node failure.

In the case of CephFS, the metadata information is kept in a separate RADOS pool, and an additional cluster of metadata servers (MDS) is needed to provide an authoritative cache for it. Frequently accessed metadata information is kept in memory by metadata servers to accelerate data access. An MDS can also give clients the permission to cache metadata locally, while guaranteeing cache coherence between them.

5.3.3 Ceph performance analysis

Wang et al. [66] evaluated the Ceph performance by measuring the I/O throughput of a realistic large-scale setup. The measurements were performed with different numbers of clients and storage nodes. The authors were able to reach 70% of the hardware capability by fine tuning the configuration of their cluster. The benchmarks were used by Ceph developers to improve their implementation in recent releases.

Van der Ster et al. [67] documented their experience with Ceph deployment at CERN IT. The benchmarks are performed on a petabyte-scale cluster, which gives a good insight into the performance of Ceph in very large setups. The metrics evaluated by this research are the bandwidth, memory consumption and rebalancing time.

Zhang and al. [151] investigated the performance of Ceph by using multiple storage benchmarking tools including *Bonnie++*, *dd* and *rados bench*. The problem with this study it that Ceph is deployed on an Openstack cloud, and the additional level of virtualization can cause an extra overhead.

Gudu and al. [71] evaluated the performance of Ceph using different setups. The scalability was evaluated by varying the number of clients and storage nodes. They proved that Ceph scales linearly when storage nodes are added. They also studied the impact of journaling on write operations and proved that placing the journal in a fast device, such as main memory, improves dramatically the overall performance of the system.

Poat and al. [72] investigated the performance of different data placement techniques like *primary affinity* and *cache tiering* on the overall performance of Ceph. They also studied the impact of using high speed SSD drives for journaling. A similar study [152] proposed non-blocking logging and lightweight transaction processing to boost the performance of Ceph with SSDs disks.

Lee and al. [70] compared the performance of the different Ceph storage backends : FileStore, KStore and BlueStore. They used a microbenchmark and a long-term workload to compare IOPS (I/O Operations per second) and WAF (Write Amplification Factor) and request latency. The Results show that Bluestore offers the best performance, which explains why it has been chosen as the default backend in recent Ceph releases.

However, all the studies reviewed so far suffer from the fact that they use benchmarking as the main performance evaluation method. This approach brings several limitations. First, the workloads used are often not representative of real world I/O patterns. Second, there is no deep understanding of the root cause of unwanted latencies, in case they happen.

Tracing can be used to provide detailed information about the Ceph behavior. Zhang and

al. proposed FSObserver to analyze the performance of Ceph, based on PCAP traces [73]. This tool observes the network traffic and filters the packets that belong to Ceph. Those packets are then decoded to know how much data is sent between the different nodes, and to compute the throughput of each node. This mechanism is limited to the network layer and does not provide any information about what is happening internally in each storage node. Recent Ceph versions are instrumented using Blkin, a userspace tracing library that enables tracking an I/O request from the time it is generated until it is fully processed [74]. It uses the Google Dapper [75] semantics to allow highlighting the different layers through which the request travels. The generated trace can be processed and visualized using Zipkin [76]. Analyzing Ceph using Blkin traces suffers from a major limitation. It is only based on high level userspace events and does not provide any information about the underlying operating system mechanisms such as disk schedulers and network queues.

In this paper, we propose a framework that uses Kernel and userspace tracing to analyze and visualize the behavior of Ceph storage clusters. The analysis provides full visibility of the complete software stack, from the userspace application down to the device driver. In the next section, the architecture and implementation of the proposed solution are detailed.

5.4 Proposed solution

In this section, we propose a methodology to analyze the behavior of a storage cluster based on trace events. The idea is to collect traces from the different nodes, synchronize them together, and provide a comprehensive visualization system that helps users evaluate the performance of the cluster.

The general architecture of our analysis is shown in Figure 5.1. The LTTng tracer is used to gather Kernel and the userspace information from the cluster nodes. If something problematic happens, the traces are collected and sent to a central machine for detailed analysis. Then, a synchronization algorithm is executed to fix the timestamps of the trace files and a data model is generated. This model is used to efficiently generate metrics and graphical views.

5.4.1 Tracing and collection triggers

Using tracing in production environments is a critical task. The overhead of tracing can be significant if a large number of high-frequency events are enabled. Furthermore, trace files usually require a lot of disk space if the target software keeps on running for a long period of time, which is the case for Ceph daemons. One possible way to reduce the tracing cost is to use a limited number of tracepoints [153], but this solution is not convenient because many

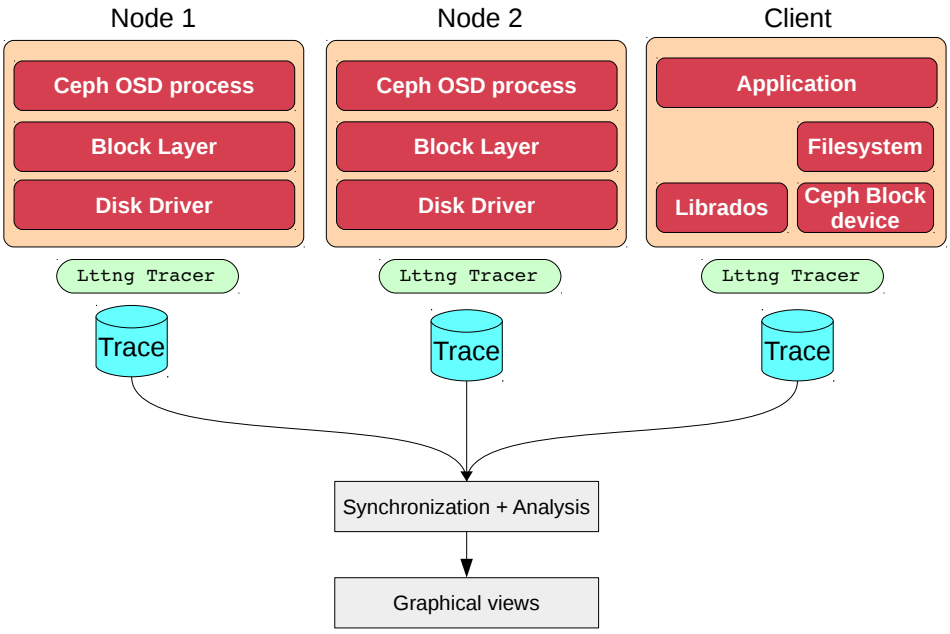


Figure 5.1 General architecture

details are needed for precise debugging.

In this paper, we use an approach based on two tracing sessions that operate simultaneously : lightweight tracing and exhaustive tracing (Figure 5.2). In the lightweight tracing session, a small number of events are traced and analyzed on the fly in order to to detect unusual behaviors, such as a slow I/O request, an OSD crash, etc. The analysis is run locally by each nodes, and the tracer output is automatically discarded after the analysis. Detection rules can be easily written by the user using a basic scripting language.

In the exhaustive tracing session, all the tracepoints required for the analysis are activated and written temporarily in-memory in a ring buffer, which reduces dramatically the tracing overhead by bypassing the cost of I/O operations required to save the file on the disk. The data comes from different layers including the application layer, the system calls, and the operating system events such as disk and network operations. When an anomaly is detected by the lightweight tracing session, a message called *collection trigger* is broadcasted over the network. This message requests the storage nodes to save the detailed in-memory trace collected by the exhaustive tracing session on disk and to send it to a central trace collection machine, where an in-depth analysis is performed. Thus, this higher data collection cost is only incurred when something significant to analyze happens, such as a long latency or crash.

After receiving the different traces, our tool correlates the events coming from the different layers and runs the in-depth analysis.

Using this approach, tracing can be activated forever without having a significant cost in terms of execution time and disk space.

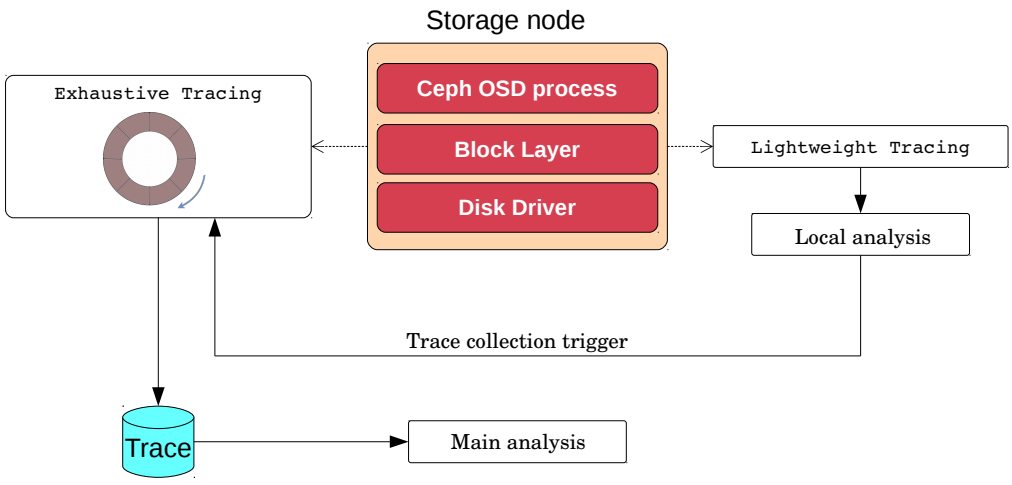


Figure 5.2 Lightweight and Exhaustive tracing

In this paper, we implemented this tracing model using LTTng [3], a low-overhead tracer available on Linux. Traces are written in per-CPU buffers to avoid the inter-CPU communication overhead and cache coherency problems between the different cores. LTTng uses RCU (Read-Copy-Update), a lock-free mechanism, to update the content of the queues without using heavy locks. The Common Trace Format (CTF) [121] is used to optimize trace writing efficiency by encoding trace events in a compact binary format.

5.4.2 Required tracepoints

To obtain detailed runtime information about the system, we use LTTng to simultaneously collect Kernel and userspace data. Userspace tracing is used to collect data from the Ceph processes (OSD, Monitors, managers, etc). Kernel tracing is used to gather information from the operating system components including the filesystem, the block layer and the disk driver. In the next paragraph, we present the different tracepoints required for our analysis.

Userspace Tracing :

Userspace tracing is important to understand the behavior of user applications. By instrumenting the key functions of the target program, we can get detailed information about the application workflow and the algorithm behind it. In order to get an overview about the cluster health, we collect data from the different Ceph processes using LTTng-UST tracepoints. The tracepoints include the storage management and the network management components (Table 5.1).

Table 5.1 Ceph userspace tracepoints

Tracepoint	Definition
opwq_process_start	The OSD process is started
opwq_process_finish	The OSD process is stopped
do_osd_op_pre	Start of an OSD operation
do_osd_op_post	End of an OSD operation
do_osd_op_pre_create	Create an object
do_osd_op_pre_delete	Delete an object
do_osd_op_pre_read	Read an object
do_osd_op_pre_write	Write the content of an object
async_enqueue_msg	Enqueuing a network message
async_write_msg	Writing a network message
osd_op_reply	Acknowledgment message received from the OSD

Kernel Tracing :

The information collected from the userspace layer provides a high-level understanding of the behavior of applications. However, computer programs are highly affected by the system environment on which they are running. In this paper, we use Kernel tracing to collect data from the different layers of the operating system.

System calls are the main boundary between the Userspace and the Kernel space. The applications use system calls to perform low-level tasks such as heap allocation, I/O operations, futex synchronization, etc. Since the main goal of this paper is to analyze storage performance, we mainly focus on I/O related system calls (Table 5.2). For every system call, we track the start and the exit time of the call, as well as the different parameters and return values.

Table 5.2 System calls events

Tracepoint	Definition
open, openat, create	Open an existing file or create a new one
close	Close the file descriptor
read, readv, pread, pread64	Read from a file
write, writev, pwrite, pwrite64	Write to a file
lseek	Reposition file offset
dup, dup2, dup3	Duplicate file descriptor

The visibility provided by the system calls is limited and insufficient to investigate low-level problems. The same system call can have different behaviors that depend on the state of the system. For instance, a READ system call can be used to read data from the disk, the network, a pipe etc. In the context of disk requests, if the required data is available on the page cache, READ gets the data and returns immediately. Otherwise, a request is inserted into the disk scheduler and the current thread until it is processed. In the context of our analysis, we need to offer higher visibility by activating lower-level events such as block layer and network events. The block layer is the part of the operating system that manages storage requests. I/O requests are fist inserted into a scheduler, deciding which one must be processed first. By instrumenting the block layer, we know exactly when a request is created by the process, inserted into the scheduler, issued to the disk and completed. The list of events that we use in our analysis are presented in table 5.3.

Table 5.3 Block layer events

Tracepoint	Definition
block_getrq	I/O request is created
block_rq_insert	The request is inserted into the elevator queue of a disk.
block_rq_issue	The request is fetched by the device driver and starts to be processed.
block_rq_complete	The request is processed by the disk and removed from the dispatch queue.

Kernel tracing can provide very low-level information about the storage hardware, which can be very useful to detect disk failures. Modern hard disks are typically connected using the SATA or the PCIe buses. The majority of disk drivers use the SCSI application programming

interface (API) as the main communication protocol. Our analysis uses the instrumentation of the SCSI interface to track the I/O requests sent on the controller bus, and if they were handled correctly. A big number of failed requests may indicate a disk failue. The tracepoints used are detailed in table 5.4.

Table 5.4 SCSI protocol events

Tracepoint	Definition
scsi_dispatch_cmd_start	the request is sent to the disk controller.
scsi_dispatch_cmd_error	the request delivery has failed.
scsi_dispatch_cmd_done	the command was handled correctly by the disk drive.
scsi_dispatch_cmd_timeout	the disk drive did not respond in time.

Ceph relies on network communication to transfer data between the different nodes. A network latency or failure can have a very severe impact on the overall performance of the cluster. By using the network layer instrumentation, we are able to collect valuable information about network packet exchanges. Those events are used by our analysis to model network communication patterns and to compute latencies. Network events required by our analysis are presented in table 5.5.

Table 5.5 Network events

Tracepoint	Definition
connect(), *accept()*, *shutdown()*, etc.	Network-related system calls
net_dev_queue	A network packet is sent
net_if_receive_skb	A network packet is received

5.5 Data Analysis and Visualization

After collecting the traces from the different storage nodes. Our tool correlates the traces and analyzes different aspects of the cluster. In this section, we present the different algorithms and data structures used by the analysis.

5.5.1 Trace correlation

Timekeeping has always been a challenge in modern processors. Technologies like TSC failed to synchronize time between the different cores, especially with variable frequency CPUs.

To solve this problem, the Kernel developers proposed the monotonic clock for fine-grained timestamping. It internally scales hardware time registers to nanoseconds while taking into consideration the variable CPU frequencies. LTTng uses this clock to guarantee total ordering of events collected from different cores, even if the wall clock is adjusted during a tracing session.

Distributed systems need complex trace synchronization mechanisms. Events are collected from different machines and it is very difficult to keep a total ordering between them. In this case, the causality between events can be used to get partial ordering, which is sufficient to have meaningful trace data. In this paper, we used the fully incremental convex hull algorithm [125] to synchronize the traces. The algorithm is based on the causality between the events of the different traces. An event from one trace must cause another event from the other trace and they must share a common key.

The network events *inet_sock_local_in* and *inet_sock_local_out* (packet sent from one node and received at the other node) are used to synchronize the different traces. Those events share the parameter *ack_seq*, which is used as a unique identifier for correlation. In Figure 5.3, we show the trace obtained before and after clock synchronization between the client and the storage machine.

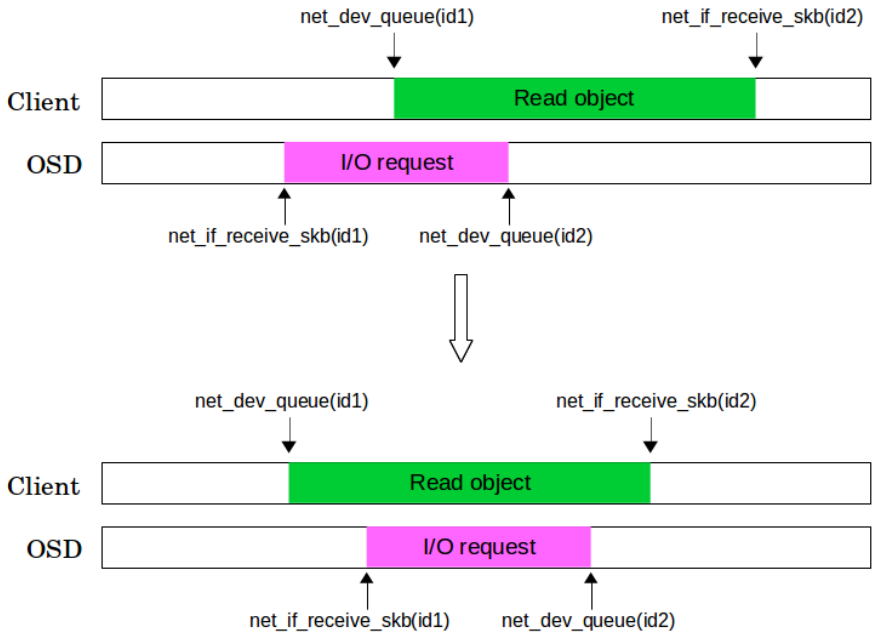


Figure 5.3 Client and OSD analysis before and after clock synchronization

5.5.2 Data model

The collected traces are usually very low-level and difficult to analyze manually. Trace events are semantically related to each other and can only be understood in the context in which they are happening. Creating an automated analysis is one of the biggest objectives of our work.

It is very important to provide enough information for the user to understand what it is happening in his system, but not so much so he would be overwhelmed by the data. One of the solutions to reduce the data size was proposed by Naser and al [154]. He proposed a taxonomy of data abstraction techniques based on pattern matching that compound the low-level events into higher level more meaningful events. This technique is more useful when the program has a sequential logic. In the case of Ceph, This technique is not very useful due to the important number of components communicating and running in parallel. Automated techniques, based on machine learning fault detection, are not very precise and generated a lot of false positives.

The methodology that we propose in this paper is based on a top-down approach. Our tools provide performance metrics that describe the overall performance of the cluser. If an issue is detected such as a slow request or an unstable throughput, the user can use more precise analysis and algorithms to find the underlying cause.

Because of the huge number of events in each trace, re-executing the different analysis each time the user selects a new time range is not a viable option. The user should be able to navigate smoothly through the trace without waiting a long time at every step. In order to avoid unnecessary computations, we decided to use a data structure in which we keep a simplified model of the system, by keeping the state of the different system components during the tracing session. Memory-based data structures like the R-tree or Segment-tree are not able to store the generated model, because the traces may be huge and the model generated cannot fit in main memory. Using relational databases is not a good choice because of their slow response time in the context of trace analysis. In this paper, we used *The Modeled State System* [155,156], a disk-based data structure that keeps the state of the system in a tree-like fashion.

We architected the model so that it keeps the state of the different machines, OSDs, I/O queues, network queues, etc. Figure 5.4 shows a summary of the state tree used in our analysis.

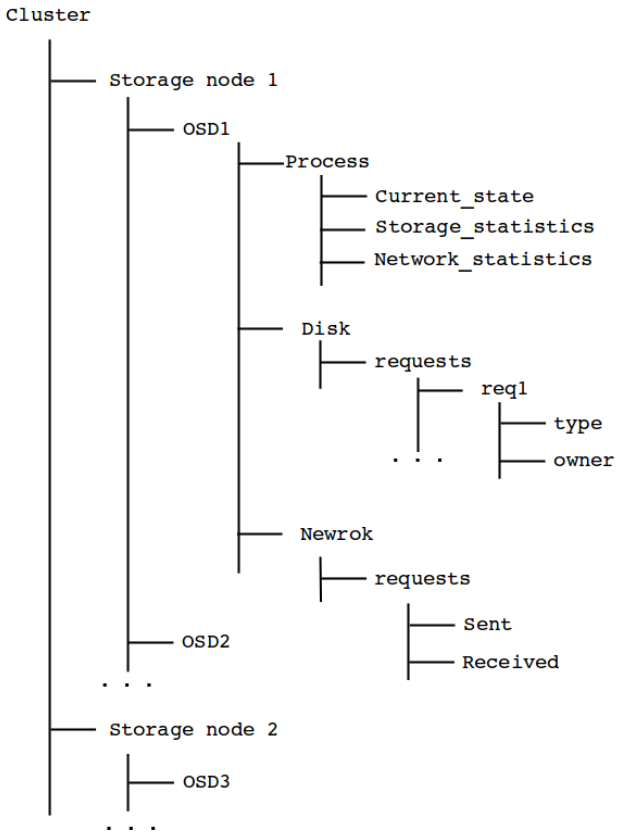


Figure 5.4 Attribute tree describing the cluster

5.5.3 Analysis module

The analysis approach used in this paper is based on a top-down approach. A good strategy is to start by analyzing high-level layers and then go into more details if a problem is detected. In the following paragraphs, we explain the different analyses provided by our tool, starting from the userspace layer down to the operating system layer.

OSD process life cycle

Every storage device is associated with a single process called OSD. This process is a daemon that runs all the time and waits for I/O requests. By tracing the OSD, we are able to know the state of the OSD throughout time. The event opwq_process_start indicates the start of the process. The OSD stays in a waiting state until a request is received. The operation is initiated and started when event od_osd_op_pre is called. An operation can be object creation, deletion, reading, writing etc, a specific event is dedicated for each one. When the request is processed successfully, the OSD goes back to the waiting state and waits for the

next operation. Figure 5.5 shows the life cycle of the OSD process.

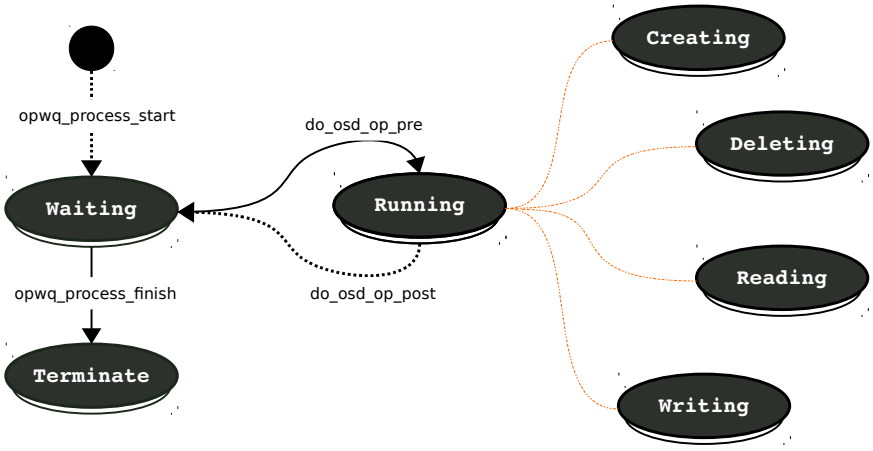


Figure 5.5 OSD daemon life cycle

Block layer analysis

Having detailed information about disk request latencies is very important to pinpoint the origin of the problem. We define three main parts : Preparation time, waiting time and service time (Figure 5.6). Preparation time is the time duration between the creation of the request and the insertion in the waiting queue. A long preparation time usually indicates that the file-system took a lot of time to create and add the I/O request to the scheduler. Waiting time is the time that the request spent waiting in the queue before being dispatched to the disk for processing. Scheduling algorithms and the number of concurrent requests are the main factors that affect the waiting time. Service time is the time required by the disk to fully process the request and it is closely linked to the hardware itself [157].

Network analysis

In our analysis, we trace the network layer communication between the nodes, in order to observe the communication patterns of the cluster and detect the nodes that generate most of the traffic. Furthermore, tracing network-related events gives us precise information about unwanted latencies. The analysis of network events is basically done with two events net_dev_queue and net_if_receive_skb, which can be correlated together using the sequence number. By looking at the header of each packet, we can know the IP address of

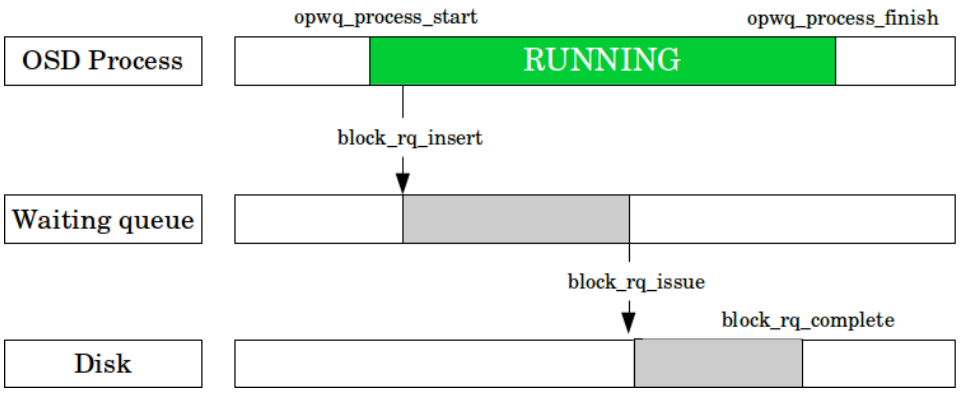


Figure 5.6 OSD I/O request details

the sender and the receiver. We iterate through the trace and we build a communication map that contains the number of packets transferred between the cluster nodes. A packet that is sent and not received is ignored by our analysis.

5.5.4 Visualization

The goal of our paper is to help users investigate the performance of their cluster with minimum effort. One of the problems of trace events is that they are very low-level and difficult to understand. Browsing the traces manually is not possible, first because the events have a very high frequency and second because some trace events can only be understood in the context in which they are triggered.

Providing a visual abstraction of the traces is a very good way to reduce the complexity of the analysis. A good visualization system must provide as much relevant information about the system as possible, while at the same time hiding unnecessary events that are useless for the specific analysis. It is important to state that an event can be useless in one context and relevant in another, depending on the aspect we are focusing on. In this paper, we developed different views for Ceph analysis and we integrated them in Trace Compass [123], an open source trace visualization tool. We decided to use Trace Compass as the main development platform because it already contains many interesting analyses that may be useful in parallel with Ceph analysis. Furthermore, all the views that we developed are synchronized with each other, which offers a great flexibility for the analysis. The user can select a time range, all the views are automatically synchronized to this time range, and all the statistics are recomputed for the same time range.

OSD activity view (Figure 5.7) shows the throughput of the storage devices supported by the cluster. The user can manually select one or many storage devices and the view shows

the activity of each, using a different color. This view is useful to have an overview on the overall activity of the cluster and to detect the most active storage devices.

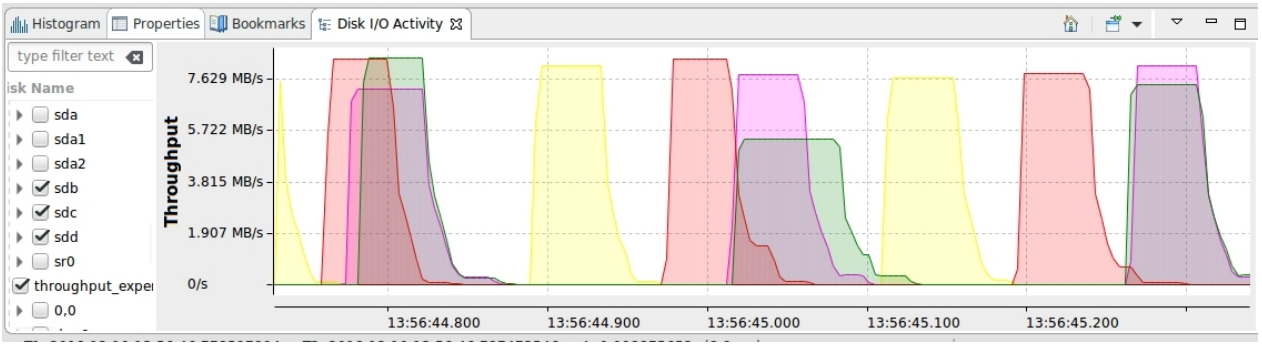


Figure 5.7 The throughput of the storage devices supported by the cluster

Ceph processes view (Figure 5.8) provides lower level details about Ceph processes. It shows on the same timeline the state of the different OSD processes using userspace tracepoints. For instance, it shows whether the OSD is IDLE or executing a disk or a network operation. Furthermore, the view uses the kernel tracepoints to show the insertion and the completion time of the I/O requests inserted by the OSD in the disk I/O queue.

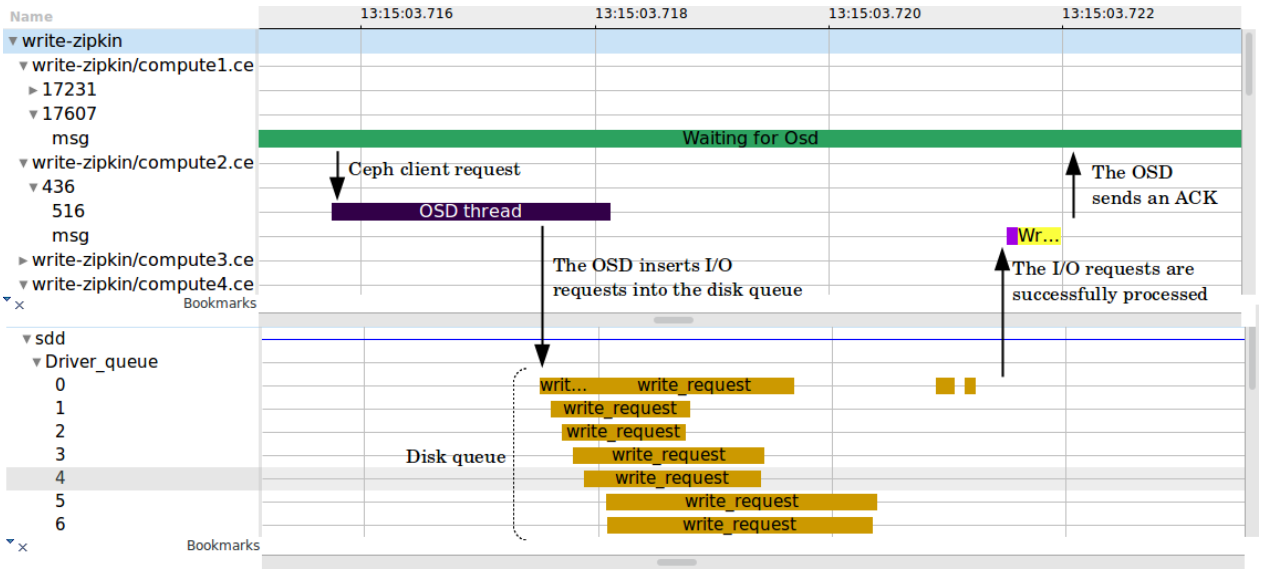


Figure 5.8 The different stages of an I/O request processing

Network view (Figure 5.9) summarizes the network exchanges that happen between the different cluster nodes. It helps the user to see communication patterns and to know the nodes

that generate more traffic. This information can be useful to choose a network architecture and hardware, appropriate to the cluster workload.

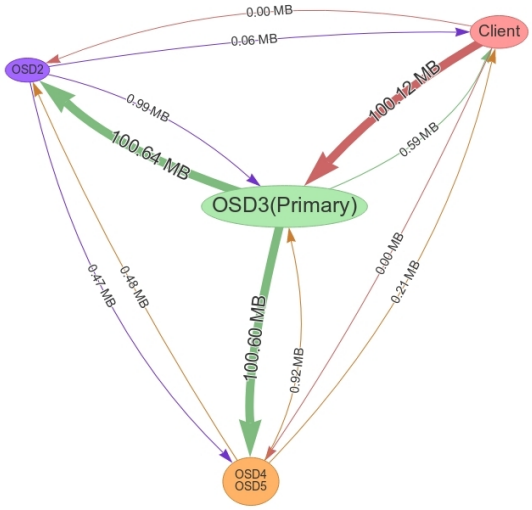


Figure 5.9 The network data flow during a write operation of a 100MB object

5.6 Use cases

Ceph storage clusters are very flexible and fit different design choices. Choosing an appropriate configuration is crucial to reach optimal performance. Tuning a Ceph cluster is a big challenge for system administrators if they are unaware of the impact that each option has on the internal behavior of the cluster. In this section, we show how our tool is able to help users first understand the behavior of their cluster and second detect performance bottlenecks caused by misconfigurations.

5.6.1 Impact of a slow disk

In this use case, we analyze a Ceph cluster that suffers from performance problems during write operations. The storage pool provides good I/O bandwidth in most cases, but sometimes write requests take longer than usual. This behavior happens rarely and only with certain objects, which makes the detection of the root cause of this problem a challenging task.

Many things can cause this unwanted behavior, such as network contention, disk contention etc. To investigate the root cause, we used our methodology to trace the cluster, synchronize the traces and analyze the interactions between the different components of Ceph. Our tool is able to provide a visual representation of the different interactions that happen between

cluster nodes, while performing an I/O workload. It shows the critical path of I/O requests from the moment they are issued by the client to the moment they are physically written on disk, which can be very helpful in pinpointing the root cause of high latencies.

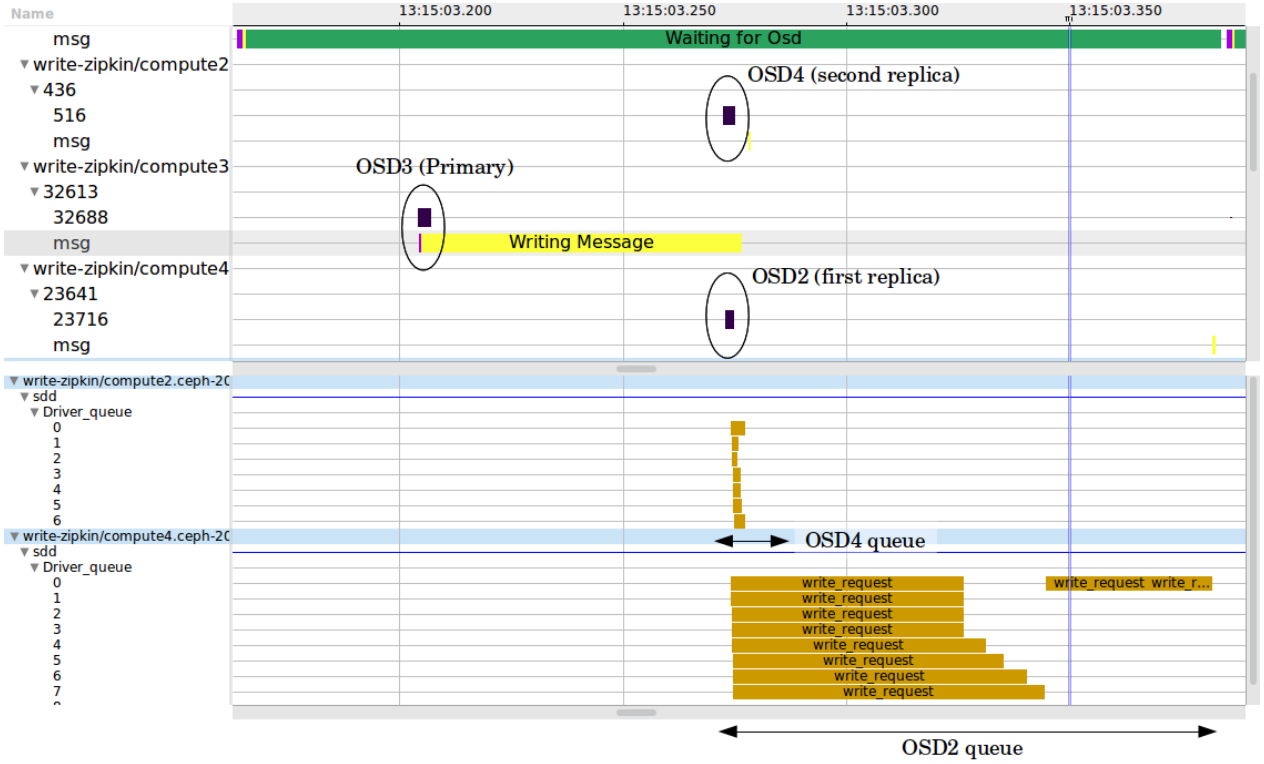


Figure 5.10 Impact of a slow disk on the overall performance of the cluster

In Figure 5.10, we see that after the client initiated the write request, the OSD3 (primary OSD) writes the data on disk and sends it to the others, OSD2 and OSD4, for replication. By looking at the waiting queues of both OSDs, we can see that the data was written very quickly by OSD4 but it took a very long time to be written by OSD2. The primary OSD does not commit the operation until the replication is successfully completed in all the secondary OSDs. As a result, if a slow disk is involved in a write operation, the client has to wait for a long time, even if all the other disks have successfully completed to request.

One way to fix such a problem is to insure that a storage pool contains disks of similar speeds, especially if we are under performance constraints. For example, if a critical application is I/O bound and requires a good I/O performance, it is important to create a storage pool that only contains SSD disks and insure that they all work properly. Otherwise, the response time of the cluster will be unstable, varying from one object to the other, and be limited by the slowest disk.

5.6.2 OSD access contention

In this section, we study the read performance of a cluster by tracking the response time of every read request issued by a client. The results have shown that most read requests take 100 ms to be processed, but some of them take around 250ms.

A big latency in a distributed storage cluster can have multiple root causes. It may come from the network, the disk, an inappropriate configuration option, etc. To have a deeper understanding of the reason for the latency, we traced the cluster and we used our tool to get a detailed latency breakdown. Figure 5.11 shows that the network and disk time is very similar in both requests. The difference is that slow requests stay in the waiting state for a long time before being executed.

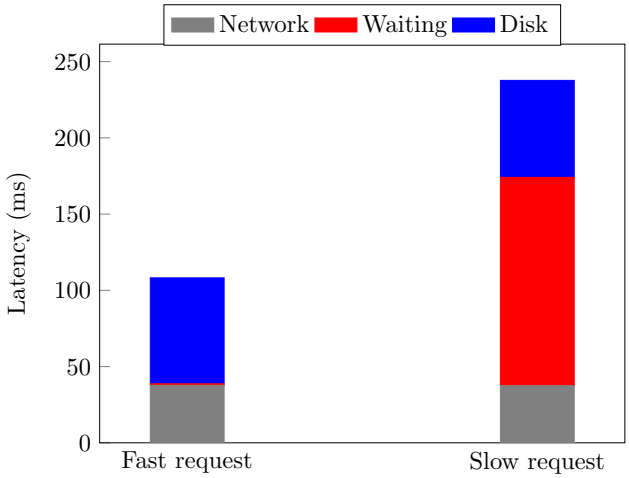


Figure 5.11 Latency comparison of fast and slow requests

We captured a detailed trace in both scenarios and used our tool to understand the differences between a slow a fast request.

In the case of fast requests (Figure 5.12), we can see that the only factors involved are the network and disks. When a read request is issued by *client1*, the OSD thread starts running and inserts I/O requests into the disk waiting queue. When the requests are processed, the OSD thread collects the data from the I/O buffers and sends it to the client through the network.

In the case of slow requests (Figure 5.13), we can see that many clients are reading the same object at the same time. The OSD doesn't handle the multiple requests in parallel to avoid compromising data integrity. Instead, the requests are serialized and executed one after the other. The figure shows that the OSD receives a read request from *client1* and it does not

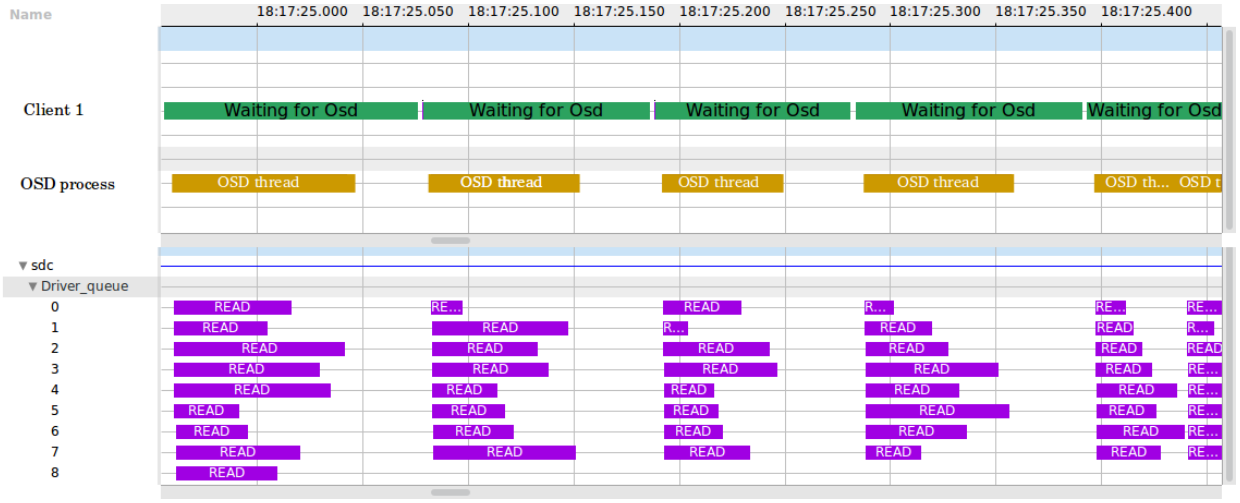


Figure 5.12 Fast request analysis

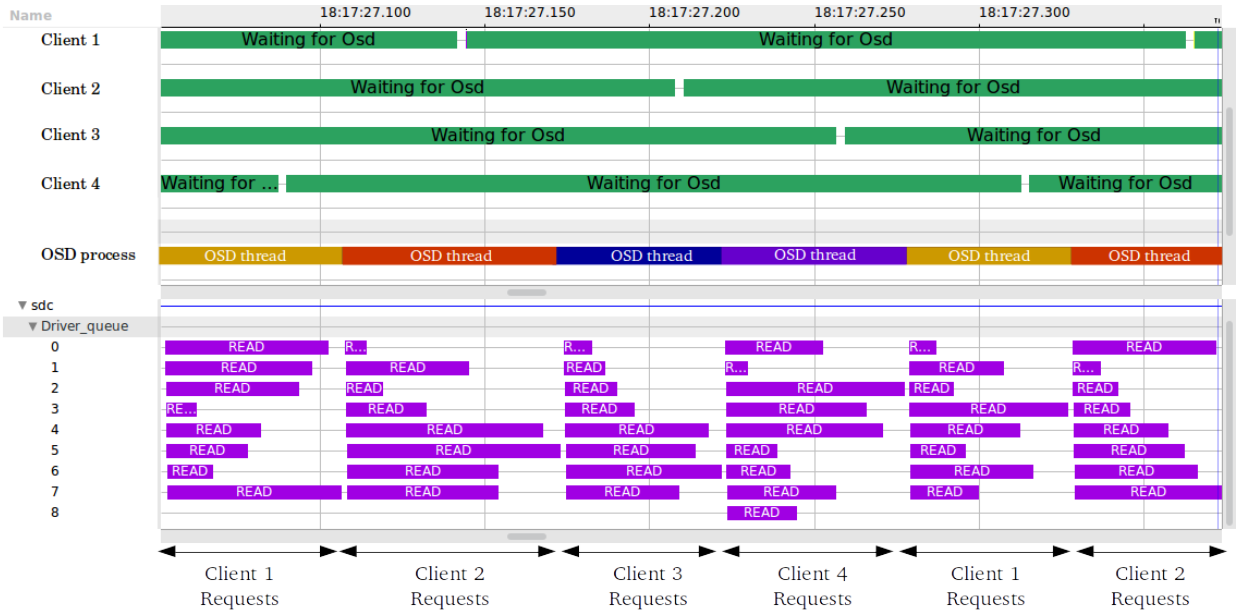


Figure 5.13 Slow request analysis

start processing the request until it finishes processing the requests of other clients, which explains the long waiting time.

An I/O request latency is affected by multiple factors and finding the root cause of the problem is very complicated without proper tools. By tracking the request from the time it is issued until it is fully processed, our tool was able to provide a latency breakdown that

shows the different phases of request processing and the timing of each phase. Using this information, system administrators are able to identify the cause of the problem and take necessary measures.

5.6.3 Data replication and OSD failure

In this use case, we want to show if a Ceph cluster is properly handling data availability and failure recovery. Our goal is to insure that the data is always safe and accessible to the users, even if a storage node goes down. The analysis will be done in two phases. First, we will see if the object is properly replicated in different machines and second we want to see what happens if the primary OSD goes down while the object is being read by a client.

We have a Ceph cluster containing multiple storage nodes and each storage node contains many disks. The replication level is set by default to 3, which means that every object is contained in three different locations at the same time.

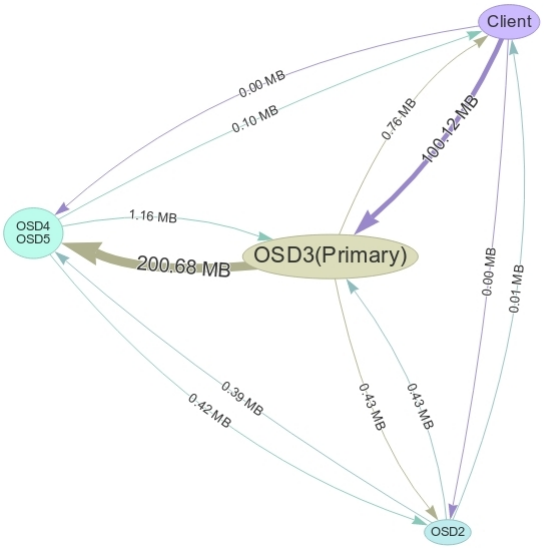


Figure 5.14 Network exchanges when two replication OSDs are running on the same storage node

Figure 5.14 shows the network communication that happens between the different storage nodes when an object of 100 MB is written by the client. The data is first sent to the primary OSD, then the latter distributes the data to the other replicas. Surprisingly, we found out that the data is sent to OSD4 and OSD5, which are running on the same machine. This behavior is not very secure because, if the machine goes down, the object is not replicated anymore and a loss of data is very likely to happen. By looking at the Ceph configuration, we

found out that the reason for that is an inappropriate configuration of the CRUSH algorithm. We fixed this problem by changing the replication type from *osd* to *host*, which means that the replication should definitely be in a separate hosts to reduce data loss risks. Another interesting observation is the fact that the same data is sent twice through the network to OSD4 and OSD5, even though both of them are running on the same machine. A more efficient approach would be to send the data only once and then write it to both disks, since both OSDs share the same main memory and can easily use inter-process communication mechanisms to exchange the data.

After we fixed the configuration problem, we wanted to see what happens if the primary OSD crashes when a client is reading data from the storage cluster. We used the *OSD activity view* to monitor the state of the different OSDs during this scenario. Figure 5.15 shows that during the down time, the client continues reading the data normally from a secondary OSD until the primary OSD becomes available again, which proves that our cluster is fault tolerant and can safely provide the data to the client, even if one of the storage nodes crashes.

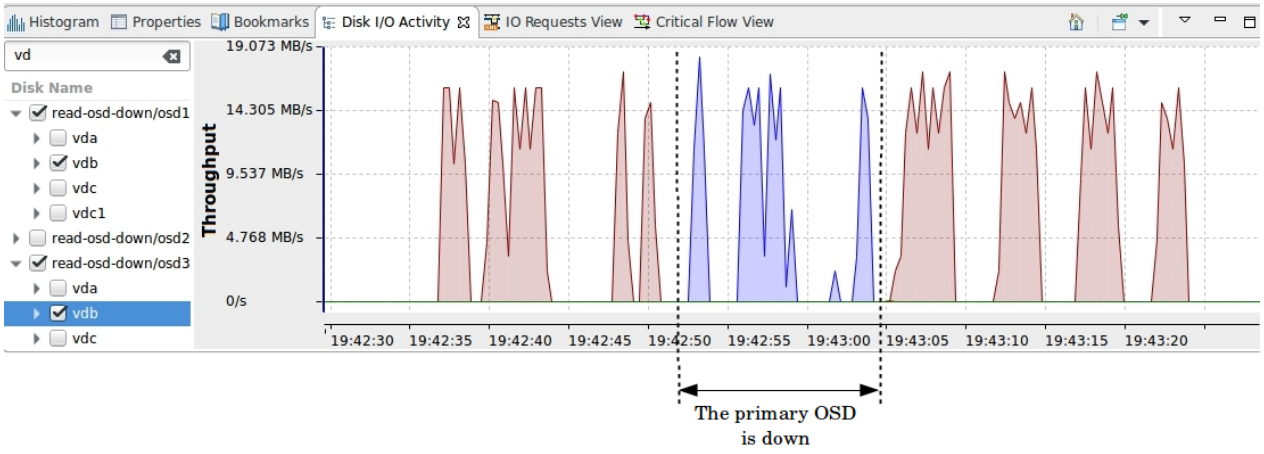


Figure 5.15 Primary OSD down then up

5.7 Evaluation

In this section, we evaluate the performance of the proposed solution. First, we want to insure that tracing does not introduce a significant overhead that may alter the normal cluster behavior. Then, we will evaluate the size of the traces and the time required to generate the data model and perform the analysis.

The Ceph cluster used for the tests is composed of 5 nodes : a client, a monitor and three

storage nodes. Each storage node contains 3 OSDs. The configuration of the storage nodes is described in Table 5.6.

Table 5.6 Hardware and software configuration of storage nodes

Hardware environment	Software environment
Processor: Intel(R) Xeon(R) CPU X5650	**Operating system:** CentOS 7.5.1804
Cores: 24	**Linux Kernel:** 3.10
Memory: 188 GB	**LTTng:** 2.10.4
Storage: 1 SSD (256 GB) + 2 HDD (1 TB)	**Ceph:** Nautilus

We used the command *rados bench* to benchmark the object storage of Ceph. We created 4 GB of data and computed the reading and writing speed of the cluster with different configurations :

— Write : Many files are written on the storage cluster with different replication sizes.
— Sequential read : Multiple clients read the files sequentially.
— Random read : Multiple clients read random chunks of the files.

Table 5.7 Evaluation of the tracing overhead of Lightweight and Exhaustive tracing

Benchmark	Replication	Base (MB/s)	Lightweight (MB/s)	Exhaustive (MB/S)	Lightweight Overhead (%)	Exhaustive Overhead (%)
Write	2	81.74	81.23	81.11	0.6	0.7
	3	62.74	62.8	62.56	0.0	0.2
	4	63.20	62.92	62.82	0.4	0.6
Sequential Read	2	113.68	113.32	113.71	0.3	0.0
	3	114.17	113.99	113.85	0.1	0.2
	4	113.98	114.05	113.59	0.0	0.3
Random Read	2	111.13	110.61	110.07	0.4	0.9
	3	112.77	111.62	111.22	1.0	1.3
	4	111.27	110.89	110.38	0.3	0.7

Table 5.7 shows that the writing speed is lower than the reading speed and that the the writing speed decreases when then replication size increases. The reason is that during a write operation, the object is copied multiple times to different disks and the client has to wait for all the replicas to be written. During a read operation, the client gets the data from the primary OSD and the other machines are not involved. Therefore, the reading performance is not affected by the replication size.

We also notice that sequential and random reads have approximately the same performance, which is explained by the fact that the stored files are striped across multiple OSDs. When

multiple clients are accessing the same file randomly, the are basically accessing different chunks of the file and therefore they are reading from different OSDs. In this case, the I/O operations are done in parallel and the throughput is similar to sequential reads.

Table 5.7 also shows that the tracing overhead does not exceed 2% with the different benchmark configurations. We notice that the overhead of Lightweight tracing is slightly lower than that of the Exhaustive tracer, but the difference is very small. We explain this by the fact that Ceph daemons spend most of their time waiting on disk and network operations, and the extra time, added by the tracepoints, is barely noticeable as compared to the waiting time. The interest of using our proposed dualistic tracing strategy is shown in Table 5.8. We see that the amount of data generated by Exhaustive tracing becomes much smaller with Lightweight tracing. For example, the trace size went from 607MB to 11MB for a write operation with a replication factor of 4. The dramatic reduction of trace size for Lightweight tracing has a direct impact on the analysis time, which goes from 63.05s to 1.19s in this case. The advantage of this approach is that storage nodes can execute the local detection algorithm without consuming a lot of resources, which is very important in production systems. The first level of detection can be executed very quickly, less than 1s for most benchmarks. If an anomaly is discovered, a collection trigger is sent and the traces recorded by the Exhaustive tracing are sent to the central analysis machine, where a more advanced algorithm can be run offline without impacting the performance of the cluster.

The scripts used for benchmarking are available in the GitHub reporsitory of the author [1].

Table 5.8 Comparison of the trace size and the analysis time of Lightweight and Exhaustive tracing

Benchmark	Replication	Exhaustive Tracing		Lightweight tracing	
		Trace size (MB)	Analysis Time (s)	Trace size (MB)	Analysis Time (s)
Write	2	414	42.83	6.8	0.74
	3	573	58.84	8.6	0.92
	4	607	63.05	11.0	1.19
Sequential read	2	39	4.33	2.5	0.31
	3	35	4.23	2.2	0.25
	4	37	4.14	2.1	0.27
Random read	2	46	5.21	2.2	0.26
	3	44	5.19	2.0	0.21
	4	48	5.41	2.1	0.22

1. https ://github.com/houssemmh/ceph-benchmark

5.8 Conclusion

In this paper, an analysis framework is proposed to evaluate the performance of distributed storage systems, using Ceph as a demonstration. Our approach consists in two phases : During the first phase, we activate a lightweight tracing session during which trace data is written temporarily in the main memory of each node. If an unusual latency is detected, the detailed traces are captured and sent to a centralized machine for post-processing. The traces are synchronized together based on the semantic causality between events, and a data model is generated to accelerate trace analysis. Different views and metrics are created to guide the debugging process.

Our tool was used to understand and debug performance issues in real storage clusters. The use cases presented cover different aspects of distributed storage systems including node failures, fault tolerance and data replication. We also proved that our tool only introduces a small overhead and can be used in production systems.

As future work, our framework can be improved by creating a distributed trace collection architecture instead of using a centralized server. Another possible improvement is to use machine learning techniques for automatic fault detection.

Acknowledgments

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, Ciena and EfficiOS.

CHAPITRE 6 ARTICLE 3 : DYNAMIC TRACE-BASED SAMPLING ALGORITHM FOR MEMORY USAGE TRACKING OF ENTERPRISE APPLICATIONS

Houssem Daoud, Naser Ezzati-jivan and Michel Dagenais
Department of Computer and Software Engineering, Ecole Polytechnique de Montreal, Montreal, H3T 1J4, Canada

E-mail : {houssem.daoud, naser-2.ezzati-jivan, michel.dagenais}@polymtl.ca

Published in IEEE High Performance Extreme Computing Conference

6.1 Abstract

Excessive memory usage in software applications has become a frequent issue. A high degree of parallelism and the monitoring difficulty for the developer can quickly lead to memory shortage, or can increase the duration of garbage collection cycles. There are several solutions introduced to monitor memory usage in software. However they are neither efficient nor scalable. In this paper, we propose a dynamic tracing-based sampling algorithm to collect and analyse run time information and metrics for memory usage. It is implemented as a kernel module which gathers memory usage data from operating system structures only when a predefined condition is set or a threshold is passed. The thresholds and conditions are preset but can be changed dynamically, based on the application behavior. We tested our solutions to monitor several applications and our evaluation results show that the proposed method generates compact trace data and reduces the time needed for the analysis, without losing precision.

6.2 Introduction

Analysing software systems is becoming exceedingly difficult and complex because of the large-scale parallelism and multiple abstraction layers. Newer systems include multiple nodes in parallel, each containing a large number of cores as well as parallel co-processors for graphics, signal processing and networking. Virtualisation, at the machine and network level, further complicates observability and performance debugging.

Dynamic analysis through execution tracing is a solution for the detailed run time analysis of

such systems. Tracing works by hooking into the different layers and locations of the software and by collecting runtime data while the software is running. It gathers valuable information about system execution and can be used in software comprehension and for finding problems and misbehaviors.

Although tracing in general is a great solution to analyse runtime behavior of systems, it may present some challenges : the trace size may be huge (in the gigabytes for only a few seconds of tracing). The large tracing data size impacts the storage space required, the analysis time, and may hinder opportunities for the timely detection of sensitive problems.

Our goal is to analyse system memory usage from the kernel point of view, since this is where actual physical resources are eventually consumed. Many state-of-the-art tools perform virtual memory monitoring from userspace [91, 158]. Their methods mostly require the instrumentation of the memory allocator of the programming language, or use a pre-loaded library that overrides the original memory allocation functions. Those solutions are not portable and can only target a specific runtime environment.

We propose a generic trace-based architecture to monitor and analyse the memory usage of any application, possibly involving multiple processes and programming languages. The challenges we face and aim to solve in this trace-based dynamic memory usage analysis method are as follows :

— The high frequency of memory operations makes for huge trace files.
— It is not usually possible to reduce the trace size by just targeting a single process, using basic trace filtering techniques, since the actual physical memory release is done out of the process context.
— Tracing can itself contribute to the memory pressure.

This paper presents a dynamic sampling technique to reduce the amount of trace data generated. The technique works by hooking on some memory related functions (i.e., using existing tracepoints) in the operating system kernel and listen to the events. Then, instead of generating events for each occurrence of the function, it only collects samples when a predefined condition is true, (e.g., a predefined but changeable threshold is passed or a particular time duration has elapsed). The proposed method :

— instruments the Kernel to get the required information ;
— provides filtering and aggregation mechanisms based on some thresholds to reduce the frequency of events ;
— generates metrics and visualizations from the trace file.

This can lead to greater efficiency when used to trace high frequency operations like me-

mory allocations. In some applications, memory related functions are called at a very high frequency, (e.g., more than 10,000 times per millisecond), generating huge trace files. As we will see later in the paper, the new approach reduces the size of trace data and the analysis time, while providing the same analysis output and precision.

The contribution of the paper lies in the proposal of an efficient architecture for in-kernel trace sampling and aggregation. The solution was tested to track high frequency kernel memory allocation functions, and the efficiency and usefulness of the method were confirmed.

In the remainder of the paper, we first review the related work. Then, after exposing the motivation for this work, we propose our new architecture and conclude with use cases and evaluations.

6.3 Related Work

Tracing is a dynamic analysis method that collects trace logs for the execution of a software system. A trace log entry can represent a function call, a system call, a signal, a network packet, etc. Unlike debugging, which is a step by step procedure going through the program execution to get its current state, or discover a problem, tracing is a more background process. Tracing collects runtime data during the execution and stores it on a local or remote disk for later analysis [159]. The tracing impact must be as low as possible to preserve the normal software behavior.

LTTng [127] is a low impact open source Linux tracing tool developed in the DORSAL Lab [1] to provide tracing capabilities for the Linux kernel and userspace applications. Kernel tracing can be performed dynamically using Kprobes, or statically using the TRACE_EVENT macro. Traces generated by LTTng can be used to analyse the run-time behavior of systems. For instance, trace-driven tools are proposed in the literature for disk block analysis [160], virtual machine analysis [161, 162], userspace level applications (e.g., Chromium browser) [163, 164], web applications [165] and live stream analysis [153]. Since LTTng is a low-impact tracer [135], and provides data at multiple system levels, it has been used in this paper to collect the tracing data about memory usage.

Griswold et al. [91] instrumented memory allocation and liberation operations of the *Icon* programming language. The instrumentation was done using macros, which was not optimized for multi-threaded applications. The trace generated was used to present the memory usage as a 2D graph where each object allocated is shown as a rectangle, proportional to the size of the allocation.

1. Distributed open reliable systems analysis lab (DORSAL) http ://www.dorsal.polymtl.ca

GCspy [166] is a memory monitoring framework developed by Printezis et al. This tool follows a client-server architecture : the collection of data is done on the server side and the visualization on the client side. The data collection requires using an instrumented memory allocator. The authors started by instrumenting the Java Virtual Machine (JVM) to track memory allocations in the Java programming language.

Cheadle et al. [158] extended GCspy to support dlmalloc, the C memory allocator used by the Glibc library. They also proposed different optimization to GCspy to reduce the communication frequency between the client and the server, as well as an automatic problem detector. GCspy uses animated images to show the state of the heap memory throughout time. This visualization is not appropriate with some applications, since the refresh rate of the animation cannot handle high frequency memory events.

Jurenz et al. [167] extended VampirTrace, a performance analysis tool, to provide a detailed memory analysis based on execution traces. Instrumentation is done using shared library pre-loading. Original memory manipulation symbols like malloc, free, calloc, etc. are overridden with new functions that contain the required tracepoints. The limitation of this method is that it targets only the processes that are started with the pre-loaded library. There is no way to trace an already running program, or to trace all the programs running on the system at the same time. TraceCompass [2] also provides a memory analysis view that uses library pre-loading to collect LTTng-UST traces.

Massif is a heap memory profiler provided with Valgrind [168]. It hooks into the application loading code using LD_PRELOAD and shows its heap memory usage. It can be used to optimize and reduce the memory usage of programs. Although Massif is a useful tool to monitor memory usage, it easily doubles the execution time of the application.

6.4 Motivation

Tracking memory usage is important to know which processes are consuming more memory resources and therefore to detect performance problems. The operating system uses complex mechanisms to manage the physical memory. When a process allocates memory, a new virtual memory space is created and assigned to the process. The real physical memory allocation is done afterward, when the memory is actually accessed by the process. The opposite operation is even more tricky. When a process asks the kernel to release (free) the memory, the kernel releases it from its virtual memory space. However, the physical release only happens when another process needs to get more physical memory.

2. http ://www.tracecompass.org

Many tools use sampling to track the memory usage of processes using the */proc* filesystem. Most of those tools use a sampling frequency of 1 Hz, which gives a low precision result. Using a fixed sampling frequency is not a good solution, since the rate of memory operations varies considerably from one process to another and through time.

Tracing is another way to get information from the kernel. If we trace memory operations like allocations and liberations, we can track in a very precise way the memory usage of processes. The problem is that those are high frequency events with the potential of overwhelming the tracing subsystem, causing lost events, which prevent precise computations. Even if we were able to collect all events using big tracing buffers, the trace file would be huge and very difficult to read and analyse.

One idea to limit the trace size is to filter the tracing events to include just one process, but this solution is not possible because, as mentioned earlier, the physical memory liberation does not always occur in the context of the target process. It may happen in the context of kernel threads or in the context of another process that reclaims the memory.

In this paper, we combine the benefits of both approaches, tracing and sampling, by proposing a dynamic trace-based sampling method. The sampling rate is defined based on the frequency of the related trace events.

6.5 Architecture

Memory management of modern operating systems is now much more complex. Each process has a contiguous virtual address space in which it allocates the required memory objects. A physical memory page is associated to a virtual one, by the page fault handler, only when the process actually accesses it.

In this paper, we provide a tool to monitor virtual and physical memory usage using a combination of tracing and sampling techniques. The proposed architecture is shown in Figure 6.1.

6.5.1 Virtual memory monitoring

The method tracks the virtual memory usage from the kernel space. It traces the different system calls related to memory allocation and release, and uses them as triggers for the Kernel Counters Reader.

Memory-related functions like malloc(), calloc(), realloc(), and free() use system calls to interact with the operating system modules where the memory management is accomplished.

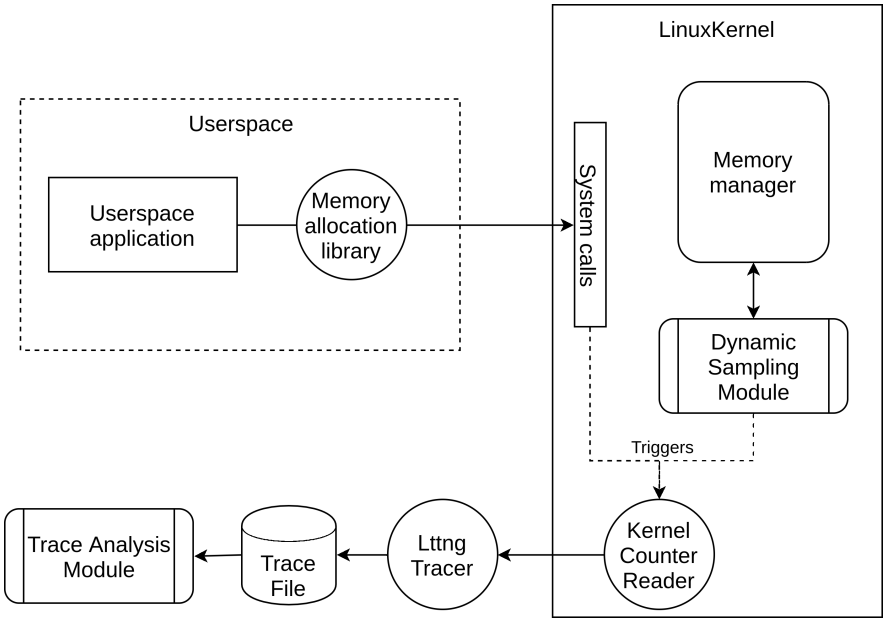


Figure 6.1 Architecture permettant de suivre simultanément l'utilisation de la mémoire physique et virtuelle

Table 6.1 shows the mapping between library functions and system calls.

Table 6.1 Mapping between memory functions and system calls

	size <= MMAP_THRESHOLD	size >MMAP_THRESHOLD
Malloc calloc realloc	sbrk	mmap
free	None, or sbrk(negative) depending on M_TRIM_THRESHOLD	munmap

The behavior of the allocator differs depending on the size of the allocation. Small allocations are achieved using *sbrk* system calls, which increase the size of an existing virtual space region. In this case, releasing the memory allocated doesn't automatically reduce the size of the virtual space. In contrast, big allocations are done using *mmap* and released right away by munmap after the memory is freed by the applications (Figure 6.2).

The virtual memory of a specific process also grows when a shared library is loaded or when a shared segment is mapped into the address space using *shmat*.

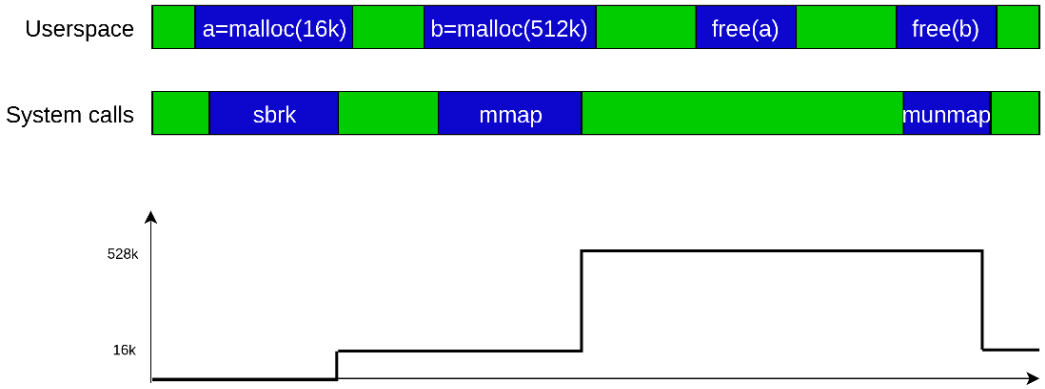


Figure 6.2 Virtual memory growth after allocation and release operations

The system calls cited in Table 6.1 are used as triggers for the Kernel Counter Reader which reads the exact value of virtual memory usage from the *mm_struct* of the concerned process and generates an LTTng trace event.

6.5.2 Physical memory monitoring : Dynamic sampling algorithm

Physical memory monitoring presents a big challenge compared to virtual memory : the memory manager of the operating system generates a huge number of events, and recording them all in a trace file is almost impossible.

Sampling can be a good solution for this case, but choosing a sampling rate is not an easy task. Some processes demand and access the memory very frequently during the execution, while others access it less frequently. A low sampling rate gives a bad precision but, on the other hand, a high sampling rate generates a huge amount of useless data for inactive processes.

In Algorithm 1, the method that dynamically changes the sampling rate, depending on processes activity, is presented.

It is a 2D sampling algorithm that uses the time and the memory variability to chose the appropriate time to get memory usage information from the Kernel data structures. An event is generated if the timer finishes, or before that if the memory variability of a process exceeds a certain threshold, as shown in Figure 6.3. The blue points represent timer events, and the red points are the events caused by the threshold.

Memory variability is computed by hooking on kmem_mm_page_alloc and kmem_mm_page_free events, which occur when a physical page is allocated or released. The Kernel keeps information about physical memory usage in the mm_struct data structure (RSS : Resident set

Input:
Sampling rate
Variability threshold

//Main thread
startTimer(*rate*)

//Timer Callback function
Function `timer_callback() :`
    processes $\longleftarrow$ *ListSystemProcesses*()
    for *process in processes* **do**
        trace_memory(*process*)
        variability[*process*] $\longleftarrow$ 0
    end
End Function

//Callback for the events kmem_mm_page_alloc and kmem_mm_page_free
Function `alloc_free_callback() :`
    process $\longleftarrow$ *getCurrentProcess*()
    if *memory page allocated* **then**
        direction $\longleftarrow$ 1
    else if *memory page released* **then**
        direction $\longleftarrow -1$
    end
    variability[*process*]$+ = PAGE\_SIZE *$ *direction*
    if *variability*[*process*] *exceeds the threshold* **then**
        trace_memory(*process*)
        variability[*process*] $\longleftarrow$ 0
    end
End Function

Algorithm 1: Dynamic sampling Algorithm

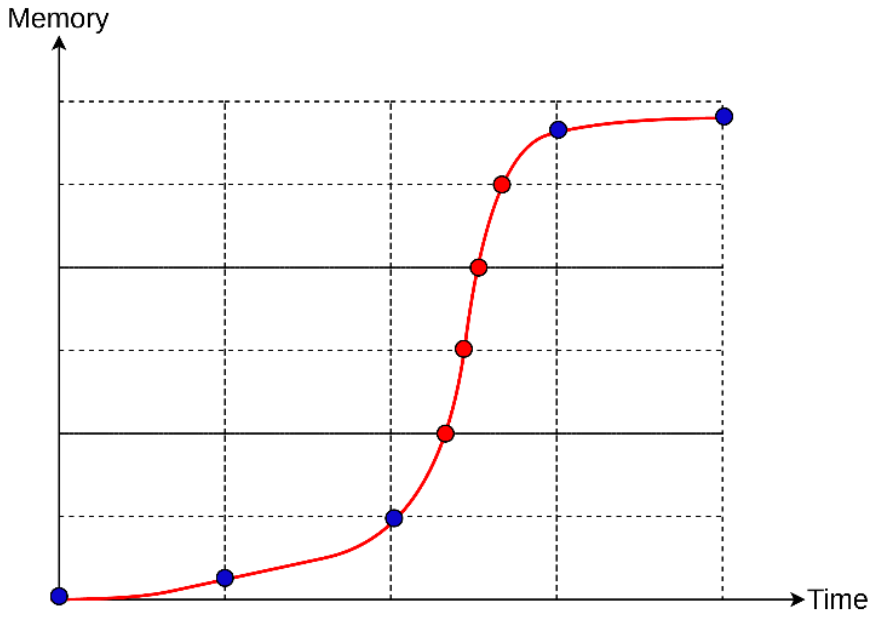


Figure 6.3 (Time, Space) Sampling

size) This counter is adjusted each time a physical page is inserted or removed from the page table of the process.

The proposed algorithm is implemented as a Kernel module and is configurable through the *proc* file system (sampling rate, variability threshold).

Lock-free data structures are used to provide a good scalability :

— RCU Hashmap is used to hold process information.

— Memory variability is defined as atomic_long to avoid using heavy synchronization mechanisms.

6.6 Evaluation

In this section, we evaluate the performance and the usefulness of our tool. Benchmarking was performed with a synthetic workload and then with real applications. We compared our method with Massif, another state-of-the-art tool, to confirm the correctness of our analysis.

6.6.1 Performance

The performance tests are executed on an Intel i7-4790 CPU @ 3.60GHz with 6 GB of main memory and an Intel SSD 530 Series 240 GB hard disk, running Linux Kernel version 4.4. The traces are collected using LTTng 2.8.

The following cases are used for benchmarking :

- *No tracing :* the program runs without any tracing mechanism.
- *Dynamic sampling :* We used our dynamic sampling module with a sampling period of 10 ms and a memory variability threshold of 10 MB.
- *LTTng all memory events :* We traced all memory allocation and release events. We used the tracepoints kmem_mm_page_alloc and kmem_mm_page_free.
- *Massif :* We used Massif, a widely used memory monitoring tool.

A program was developed to generate a memory access workload. It allocates, accesses and frees a memory buffer of a certain size, and repeats the operation until it reaches 20GB of workload. The execution time of this program with the different configurations is presented in Table 6.2 and Figure 6.4. The same benchmark is also performed with real applications : Firefox, a widely-used web browser, and Totem, the default movie player of the GNOME desktop. The results are reported in Table 6.3.

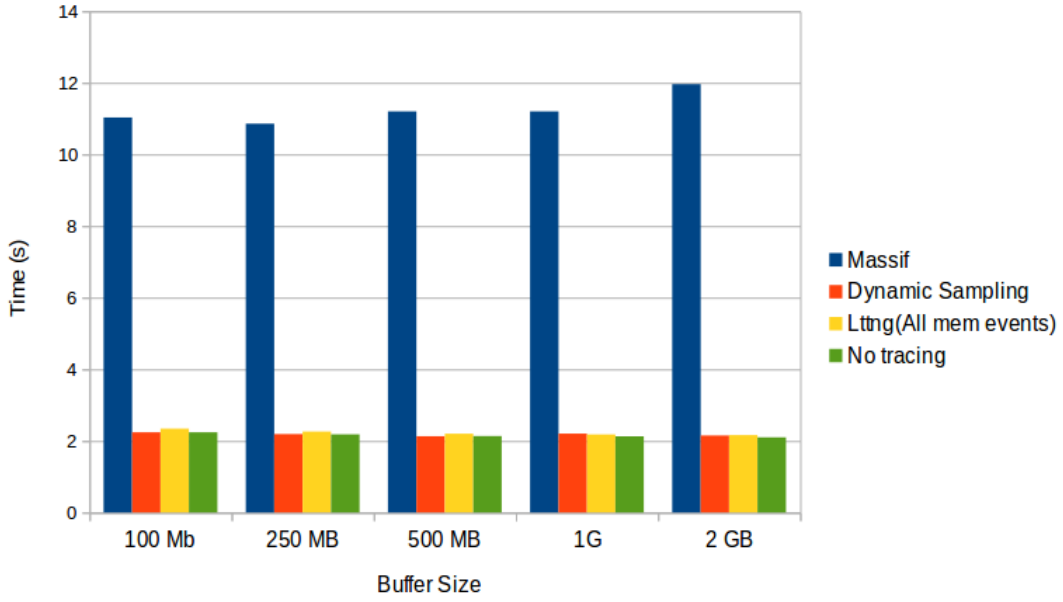


Figure 6.4 Tracing impact on execution time

The results show that the overhead of Massif is very high, as compared to other cases. It is 5x slower with the benchmarking program, 3x slower with Totem and 20x slower with Firefox. In contrast, the overhead of the two other cases is almost negligible. It doesn't exceed 1% in all cases.

It is expected that the Dynamic sampling algorithm and *LTTng all memory events* are similar in terms of execution time because we are tracing the same kernel functions in both cases, the difference is in the number of events generated. Table 6.4 shows that with a sampling

Table 6.2 Execution Time in seconds as a function of the Malloc buffer size, with different tracing mechanisms

Malloc Buffer Size / Tracing Mechanism	Massif	Dynamic Sampling	LTTng (All memory events)	No tracing
100 MB	11.03	2.24	2.34	2.239
250 MB	10.86	2.19	2.26	2.183
500 MB	11.2	2.13	2.2	2.134
1G	11.2	2.203	2.18	2.126
2 GB	11.96	2.15	2.16	2.097

Table 6.3 Execution Time in seconds for some applications with different tracing mechanisms

Application / Tracing Mechanism	Massif	Dynamic Sampling	LTTng (All memory events)	No tracing
Firefox	51	2.509	2.59	2.51
Totem (10 seconds video)	28	10.641	10.645	10.752
Benchmark application(Buffer size=500MB)	11.2	2.13	2.2	2.134

period of 10 ms and a memory variability threshold of 10 MB, we can reduce the size of the trace by a factor between 3 and 7 for a normal workload.

An interesting phenomenon happens when the buffer size is more than 4 GB. The operating system goes into a thrashing state. Memory pages start to be moved between the main memory and the swap space, which creates a huge memory activity. Tracing all memory activity at this point is very inefficient, and somehow impossible since the number of events is very high. The Dynamic Sampling Mechanism is able to handle this case by filtering the unnecessary events on the kernel side.

Table 6.4 Numbers of events in the trace file generated by LTTng (All memory events) and the Dynamic Sampling Mechanism

malloc size / tracing mechanism	LTTng (All memory events)	Dynamic Sampling	Reduction factor
5 GB	8508992	142984	59.51
4 GB	3636861	52771	68.92
2 GB	102387	29686	3.45
1G	101200	28317	3.57
500 MB	124644	28425	4.39
250 MB	173558	32834	5.29
100 Mb	300623	40635	7.4

6.6.2 Correctness

In this section, we use our tool to monitor the virtual and the physical memory usage of different applications and we use Massif, the Valgind memory profiler, to validate that the results are the same with both tools.

At first, We traced our program using the dynamic benchmarking mechanism and we used TraceCompass to show the results graphically. The output of our tool (Figure 6.5) corresponds perfectly to logic behind the code. The program allocates 500 MB, access the allocated memory and the frees it.

We can see that the virtual memory, shown in red, is allocated when malloc() is called. The physical memory, shown in blue, is allocated when the memory pages are accessed using memset(). The virtual and physical memory are released during the free() function call.

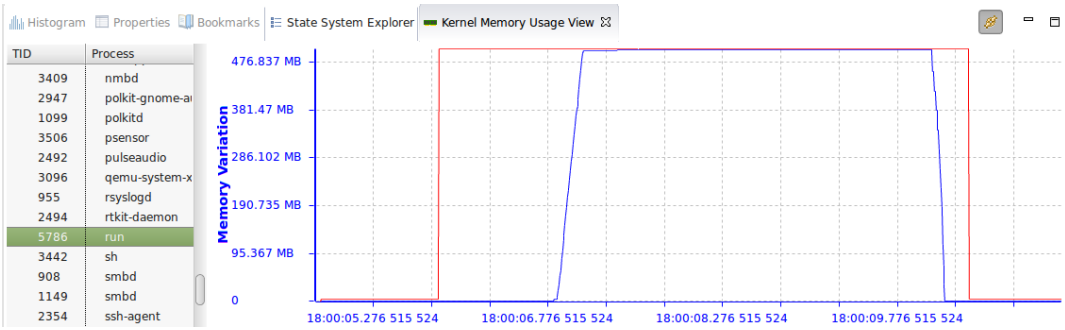


Figure 6.5 Virtual and physical memory usage monitoring

Figures 6.7 and 6.6 show that both tools give similar memory usage graphs for Firefox. Totem memory usage is also plotted by both tools in Figures 6.8 and 6.9 which displays the same output for both approaches.

Our tools bring other important advantages when compared to Massif. Physical memory usage is shown and the analysis covers all the processes running on the system at the same time, not only one targeted process.

6.7 Conclusion

In this paper, a framework to collect memory usage information for enterprise applications is proposed. It includes a dynamic sampling algorithm to gather runtime information from the operating system kernel. The method checks if a certain time has elapsed or if a threshold was reached and then gathers information from the kernel data structures and generates trace events to be processed and analysed later. The thresholds are dynamic and can be updated

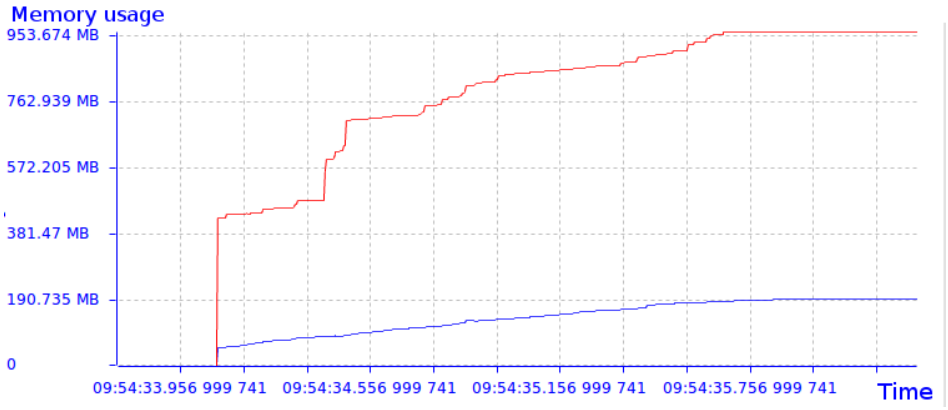


Figure 6.6 Firefox memory usage at startup using LTTng

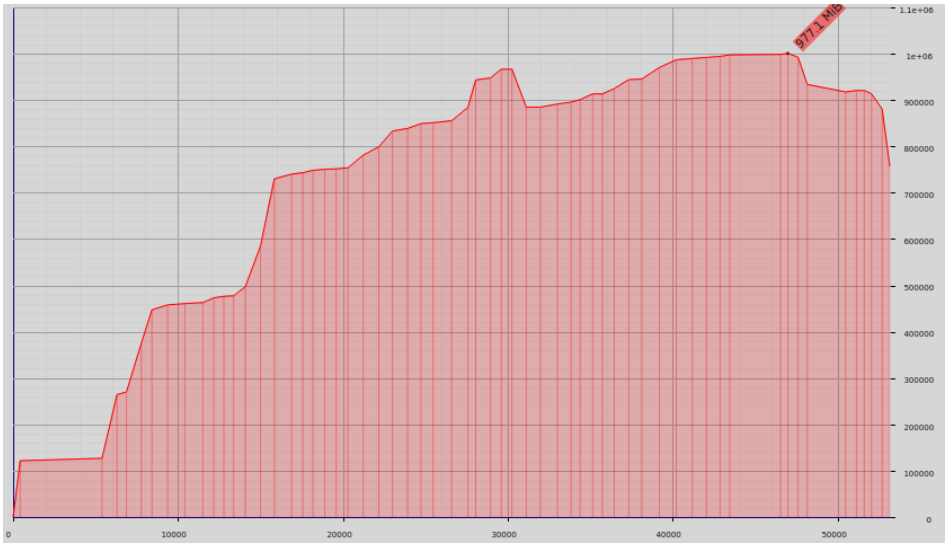


Figure 6.7 Firefox memory usage at startup using Massif

based on the application behavior and memory usage pattern (e.g., the rate of memory allocation calls).

We have tested our method against some real world applications like Firefox and the Totem video player, and the results demonstrate that the performance cost of the proposed approach is negligible while the precision is preserved.

The proposed solution was used to analyse memory usage. However, the architecture is generic enough to be used for any other resource usage metric. It can actually be used for other high frequency tracing events within the operating system kernel, like network usage, disk I/O, etc. Extending the proposed method to support other kinds of metrics, and using other high frequency events, will be investigated as a future work.

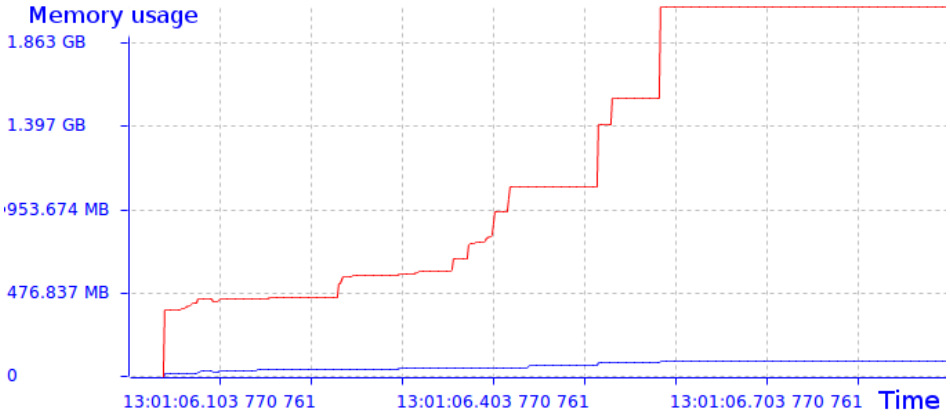


Figure 6.8 Totem memory usage to play a video using LTTng

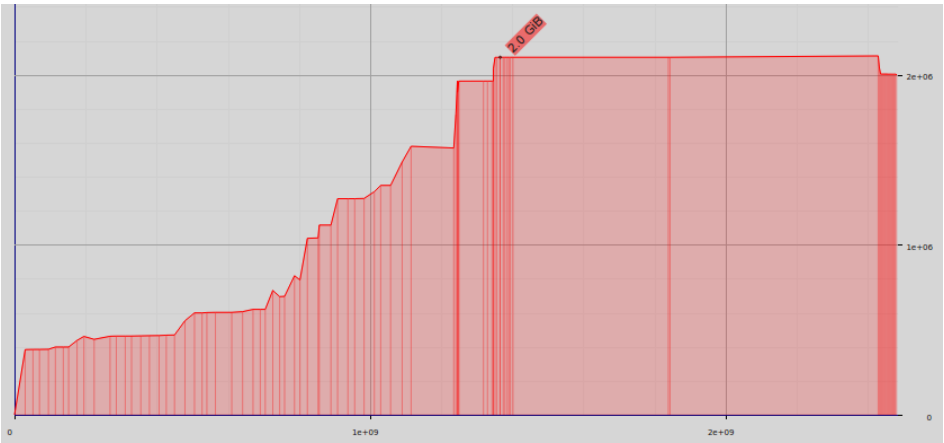


Figure 6.9 Totem memory usage to play a video using Massif

Acknowledgments

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson and EfficiOS for funding this project.

CHAPITRE 7 ARTICE 4 : MULTILEVEL ANALYSIS OF THE JAVA VIRTUAL MACHINE BASED ON KERNEL AND USERSPACE TRACES

Authors

Houssem Daoud and Michel Dagenais
Department of Computer and Software Engineering, Ecole Polytechnique de Montreal, Montreal, H3T 1J4, Canada

E-mail : {houssem.daoud, michel.dagenais}@polymtl.ca

Submitted to Journal of Systems and Software

7.1 Abstract

Performance analysis of Java applications requires a deep understanding of the Java virtual machine and the system on which it is running. An unexpected latency can be caused by a bug in the source code, a misconfiguration or an external factor like CPU or disk contention. Existing tools have difficulties finding the root cause of some latencies because they collect data mainly from the userspace level, which doesn't provide enough details about the system on which the application is running. In this paper, we propose a multilevel analysis framework that uses Kernel and userspace tracing to help developers understand and evaluate the performance of their applications. Kernel tracing is used to gather information about thread scheduling, system calls, I/O operations, etc. and userspace tracing is used to monitor the internal components of the JVM such as the garbage collectors and the JIT compilers. By bridging the gap between kernel and userspace traces, our tool provides full visibility to developers and helps them diagnose difficult performance issues. We show the usefulness of our approach by using it to detect problems in different Java applications.

7.2 Introduction

Computer programs are becoming increasingly complex. Modern applications involve different interdependent components interacting together and running on optimized multi-core processors. In recent years, there has been an increasing interest in virtual machine based programming languages, because of their flexibility and ease of deployment. A well known example of these languages is Java. Java has been considered as one of the most widely

used programming languages in industry [169]. It is platform independent and it offers a great amount of advanced technologies like automatic memory management and Just-in-Time compilation.

However, the advanced mechanisms provided by Java made performance debugging more challenging. The developer is usually not fully aware of the internal behavior of the Java virtual machine (JVM) and, as a result, he is unable to pinpoint issues that may happen at runtime. For example, a misconfigured option can cause unnecessary garbage collection cycles that can be easily avoided once detected.

Unexpected behaviors are very hard to detect without proper tools. A wide range of analysis tools are available for developers to evaluate the performance of Java applications. Those tools may be classified into two main classes : monitoring tools and profilers. Monitoring tools provide high level information about the current state of the virtual machine. They present statistics about object allocation, garbage collection, active threads, etc. Profilers give additional information about the logic of the application. By periodically collecting the call stacks of the different threads, profilers can detect slow functions and the logic of the code behind them.

The problem with existing tools is that their source of information is limited to the userspace level. Operating system mechanisms such as CPU scheduling [170] and I/O request management [126] have a huge impact on the overall application performance, and yet they are not covered by traditional analysis tools. On the other hand, Kernel-level monitoring is insufficient because it provides very low-level details about the system but fails to describe the current state of the application [154].

In this paper, we propose a unified Java performance analysis framework that covers the whole software stack, from the user application down to the operating system. We achieve that by using a hybrid approach based on kernel and userspace tracing [127] [171]. We use Kernel tracing to collect low-level information from the operating system (CPU scheduler, Block devices, network interfaces, etc) and userspace tracing to collect information from the different components of the Java virtual machine, such as the garbage collector and the JIT compiler. The traces are collected, synchronized and analyzed using a correlation model. By collecting data from multiple sources and layers, our tool offers full visibility of the system and helps in detecting problems that are very difficult to see otherwise.

We presented different use cases where our tool was able to detect low-level latencies that are difficult to analyze using traditional tools such as I/O and CPU contention. We also proved that our tool doesn't introduce a big overhead and can be used without altering the normal behavior of applications.

The main contributions are as follows :

— Instrumentation of the Hotspot [172] virtual machine using LTTng [3], a low-overhead tracer available in Linux.

— Multilevel performance analysis model that correlates and analyses trace data from different sources.

— A Visualization system that helps users understand and identify performance degradations.

The rest of the paper is organized as follows : in section 7.3, we introduce the different components of the JVM and we discuss the existing Java performance analysis tools. Then, we describe the architecture of the proposed solution in sections 7.4 and 7.5. We present three use cases in section 7.6 and we evaluate the efficiency of the framework in section **??**. The conclusion and future work are then presented.

7.3 Background

In this section, we provide a description of the different components of the Java Virtual Machine and we list the different studies related to the performance analysis of each. Then, we list some existing Java performance analysis tools and we discuss their limitations. Finally, we define tracing and we justify the choice of Lttng as the main tracer for our framework.

7.3.1 Java Virtual Machine Architecture

The Java programming language was designed with the purpose of writing platform independent programs that can be run on any hardware architecture. To meet this goal, the idea was to have an intermediate software layer able to interpret and execute Java programs, the Java Virtual Machine (JVM).

There are many implementations of the JVM developed by different organisations. The most popular ones are Oracle Hotspot [173], Oracle JRockit [174], and Eclipse OpenJ9 [175]. Those JVMs satisfy the specification published by Oracle in order to ensure interportablility [176]. This abstract specification does not provide implementation details. Developers have the freedom to introduce different optimization mechanisms into their virtual machines. Hotspot is an open source JVM and is distributed with the OpenJDK package. It is considered as the reference implementation, and will be used throughout this paper.

In the next sections, we will describe the general architecture of Hotspot. We focus on the performance aspect of memory management, JIT compilation and thread management.

Memory management : Garbage collection

In more traditional programming languages like C and C++, memory management operations, like allocating and releasing objects, have to be called explicitly by the programmer. The advantage of this approach is that it gives the developer full control over the behavior of the program. However, manually managing memory references is a delicate task and may be the source of different programming errors. The most common problems are dangling pointers and memory leaks. Many studies have focused on helping programmers to detect and avoid those problems in languages with explicit memory management [177–180]. However, those techniques require a lot of care and rigor from programmers. To free developers from the burden of memory management, many programming languages like Java and C# introduced automated mechanisms based on garbage collectors [181]. The main tasks of a garbage collector are allocating memory objects and automatically releasing them when they are no longer needed. The collection process starts when a certain condition is met, usually when the heap occupancy reaches a certain threshold.

Designing a garbage collector implies different trade-offs. The first trade-off concerns collection frequency. Garbage collection should be executed frequently enough to ensure that unreachable objects do not stay in memory for a long time, but on the other hand infrequently to avoid slowing down the application. The second trade-off is between response time and space efficiency. When the program allocates an object, the garbage collector must be able to quickly find space for the new object, and at the same time avoid memory fragmentation [80] by finding the best fit for the object.

Java virtual machines propose different mechanisms to improve the performance of garbage collectors. Compacting collectors reduce fragmentation by periodically scanning the heap and moving the used blocks to an adjacent place [182]. Parallel collectors use multiple threads to execute garbage collection tasks in a concurrent manner [183, 184]. Generational Garbage collectors divide memory objects into different generations based on their age and set different collection parameters for each [185].

Analyzing the impact of garbage collectors on applications behavior has always been a subject of interest among researchers. Studies [110–112] used benchmarking to compare different garbage collector implementations. Many memory allocation workloads are executed with different GC configurations in order to compare their impact on execution time. Lengauer and al [113] provided a comprehensive study of the different benchmarking suites in order to help the user choose a convenient workload for their tests.

The major drawback of benchmarking is that synthetic workloads are limited and cannot exactly imitate real world applications. Many studies proposed to compute objects lifetimes based on garbage collector traces [114–116]. Collecting garbage collection events solely does not provide enough context about the application. To deal with this problem, Ricci and al. proposed *Elephant tracks* [117], a tracing tool that traces the entry and exit of functions in addition to GC events. The advantage of this tool is that it gives more context about the application when a garbage collection happens. However, tracing all functions introduces a big overhead. The same result can be achieved by recovering the call stack when a garbage collection is triggered, as we describe in details in paragraph 7.4.1.

Just-In-Time (JIT) Compilation

The Java virtual machine uses the Runtime environment to interpret the bytecode generated by the Java compiler. If a certain method is called multiple times, the JVM may decide to compile it to more optimized native code, in order to improve the execution time. This process is called JIT compilation.

Compilers optimize the code using techniques like inlining, loop unrolling, branch prediction, etc. Two types of compilers are available. Client compilers (C1) are designed for client-side applications. Those compilers are fast and generate decently optimized code. C1 compilers are designed to work on low resource machines and do not slow down the startup time of the application. On the other hand, servers compilers (C2) are designed for server applications and provide highly optimized code. C2 compilation is slow and using it during startup introduces a significant overhead. Recent JVMs provide a technique called *tiered compilation* [186], which combines both compilers. A hot method is first compiled by the C1 compiler and then recompiled by C2 if it is executed more often than a certain threshold. The advantage of tiered compilation is that it does not slow down the startup time of an application, but at the same time provides highly optimized code for very hot methods.

Many studies have been conducted to evaluate the effects of compilation on the overall program performance. Selective compilation is mainly based on the observation that the program spends most of the time executing the same functions, which we call *hot functions* [187, 188]. The biggest challenge is to set the right threshold for selecting those functions. A high threshold makes the compiler very conservative, declining many optimization chances. But setting a low threshold can trigger a large number of unnecessary compilations. Finding the ideal number is usually based on predictions based on runtime tracing, usually using performance counters [189].

Thread Management

Thread management is a very important task of the Java virtual machine. Different types of threads are involved. Some are created by the Java application itself and other threads are needed by the JVM to execute low-level operations like signal handling and garbage collection. The Hotpot virtual machine uses a 1 :1 threading model. Every JVM thread is associated to a native operating system thread.

The JVM defines different tread types. The most important ones are Java threads, GC threads, compiler threads and VM threads (Figure 7.1).

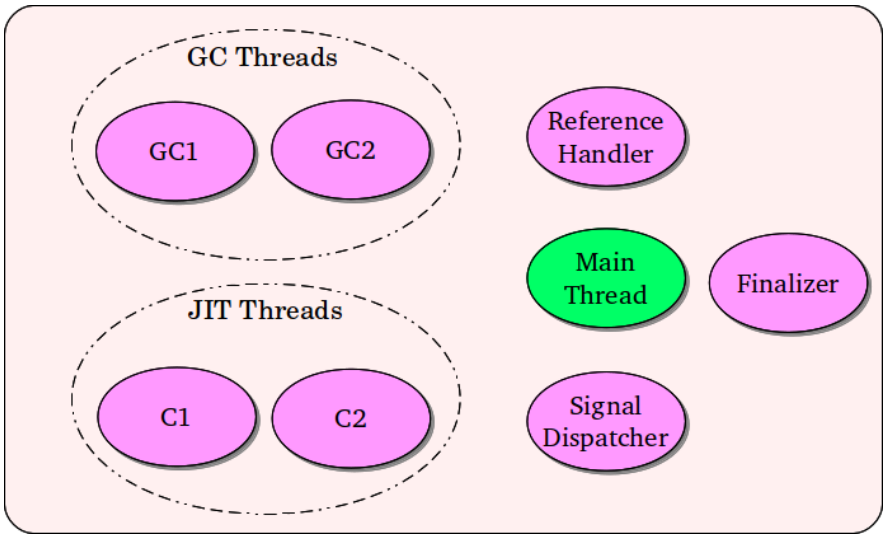


Figure 7.1 Overview of the different types of threads

The possible states of a thread are presented in table 7.1.

Table 7.1 Thread states

State	Defintion
thread_new	the thread has been created, but not yet started
thread_in_Java	the thread is currently running Java code
thread_in_native	the thread is running native code
thread_in_vm	the thread is executing virtual machine operations
thread_blocked	the thread is blocked, because of a lock, a disk operation, etc.

7.3.2 Java performance analysis tools

Analyzing the performance of Java applications is one of the biggest challenges due to the complexity of the JVM and the different components involved. Many technologies are provided to help programmers detect performance issues which can be caused by inefficient programming or a non optimal configuration.

Java Management Extension (JMX) [190] is a performance monitoring technology that helps developers manage the different resources of an application locally or over the network. Every monitored resource is described by an MBean such as ThreadMXBean, ClassLoadingMXBean, MemoryMXBean, etc. Internally, an MBean is a Java object and it can be registered dynamically at runtime. For example, a Memory management MBean contains attributes like *total_memory* and *free_memory* and methods like *getHeapMemoryUsage()*. MBeans are managed by an MBeanServer which can be controlled using different connectors like RMI, JMXMP, or any other user-defined protocol. JMX Beans are developed in Java and, as a result, they usually introduce a large overhead and can alter the normal behavior of the program due to heap pollution.

The Java Virtual Machine Tool Interface (JVMTI) [191] is a programming interface to inspect and to control the execution of Java applications. It is used by diagnosis and monitoring programs like debuggers, profilers, memory analyzers, etc. The clients of the JVMTI are called *agents*. An agent can send requests to the JVMTI to execute control functions. For example, it is possible to force a garbage collection, access and modify a Java class attribute, etc. The agent can also receive a notification from the JVMTI if a certain event is triggered, such as thread creation, class loading or a JIT compilation. JVMTI also allows *bytecode instrumentation*, the ability to alter the normal behavior of the application by injecting a new code at runtime. This can be useful for instrumentation or to update execution counters.

Many monitoring tools have been developed using the technologies described above to help the developer gather valuable information at runtime. JConsole [192] is a graphical tool that uses JMX to provide basic information about threads, memory, classes etc. It also allows data collection from user-defined MBeans, which provides a more personalized analysis. VisualVM [193] is another monitoring tool similar to JConsole, but with more advanced views. It offers the possibility to track the state of each thread during the execution, evaluate memory usage and provide heap dumps.

When it comes to profiling, existing tools like jstack, JProfiler and HPROF [194] periodically call the function *GetAllStackTraces* provided by JVMTI, to recover the call stacks, and use the collected data to evaluate function durations. The problem with this technique is that

the function *GetAllStackTraces* only returns when all the threads reach a safepoint, which affects the sampling period. This limitation is known as *safepoint-bias*. HonestProfiler [195] solves this problem by using an undocumented API function *AsyncGetCallTrace* to sample the Java application. *AsyncGetCallTrace* captures exactly what a JVM is doing, without waiting for a safepoint. The only drawback of this technique is that it only captures the call stacks of Java threads and other virtual machine threads like GC or JIT threads.

The existing monitoring tools provide a good view for what is happening in the application, but only from the userspace level. There are many other factors that have a big impact on the performance which cannot be seen by those tools. For example, an application can be affected by a slow disk drive or CPU contention. This kind of information can only be obtained at the operating system level, not from the userspace level. In the rest of the paper, we show how synchronizing kernel traces with userspace information can provide a comprehensive view of what is really happening in the system, and help detecting issues otherwise very difficult to see by traditional tools.

7.3.3 Tracing

Tracing is a mechanism that collects execution data from an application at runtime. In order to avoid the observer effect, tracers use advanced techniques to minimize the overhead. Therefore, tracing is one of the most used performance analysis techniques for production systems. Instead of stopping the application, the tracer writes the different events into a file, and the analysis is executed offline. A trace event is described by a name, a timestamp, and a payload containing multiple parameters. It is possible to trace system calls, interrupts, function calls, etc.

LTTng is an open source tracing tool developed at the DORSAL laboratory, as an improvement to the LTT tracer. It offers low-overhead kernel and userspace tracing for the Linux operating system. The Linux kernel can be instrumented statically using TRACE_EVENT macros, or dynamically using Kprobes [138]. The TRACE_EVENT macro has been adapted by Lttng to userspace tracing. The developer can instrument any application by defining personalized tracepoints and inserting the required probes in the source code.

LTTng uses per-CPU circular buffers and lock-free data structures like RCU to support highly parallel applications. Read-Copy-Update (RCU) is a synchronization mechanism that allows reads to happen concurrently with updates, while at the same time avoiding cache coherence overhead between CPU cores. Trace events are written in circular buffers using The Common Trace Format (CTF), a binary format designed for trace files, more efficient than text formats used by loggers [121].

7.4 Proposed solution

Analyzing the runtime behavior of a Java application requires a deep understanding of the JVM and the environment on which it is running. To have full visibility of the application, our approach is based on collecting data from multiple layers : the user application, the Java virtual machine and the operating system. The collected traces are then synchronized and analyzed by inspecting the occurrence and the causality of the different events. The general architecture of the proposed analysis framework is described in Figure 7.2.

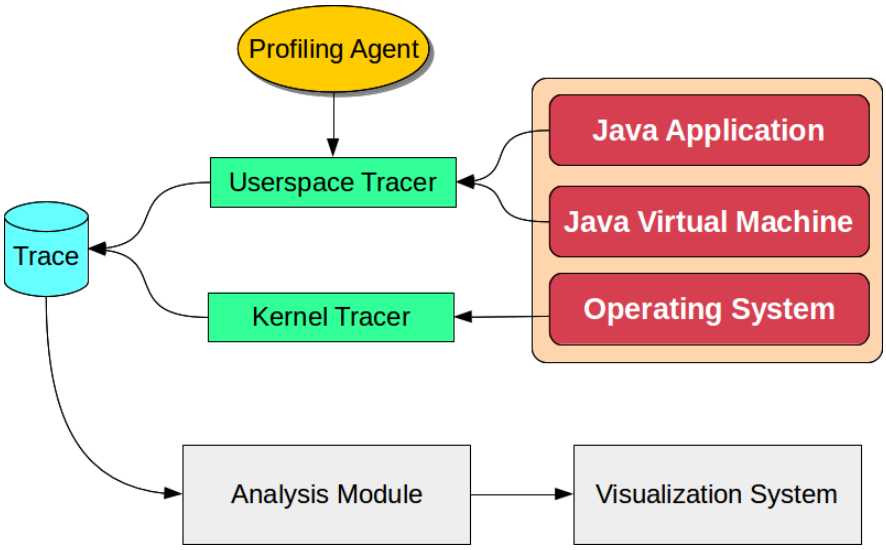


Figure 7.2 General Architecture

7.4.1 Data Collection

Userspace Tracing

Lttng-UST is able to collect data from userspace applications. The data collection is achieved by inserting probes in the source code of the application and receiving an event every time the execution hits that point. The event payload contains the data required by the analysis. In order to understand the runtime behavior of Java applications, we instrumented the different components of the Java Virtual Machine. Figure 7.3, summarizes the different userspace tracepoints used for the analysis

The trace events used can be organized into different categories :

— Java threads tracepoints (Table 7.2)
— Garbage collector tracepoints (Table 7.3)
— JIT compiler tracepoints (Table 7.4)

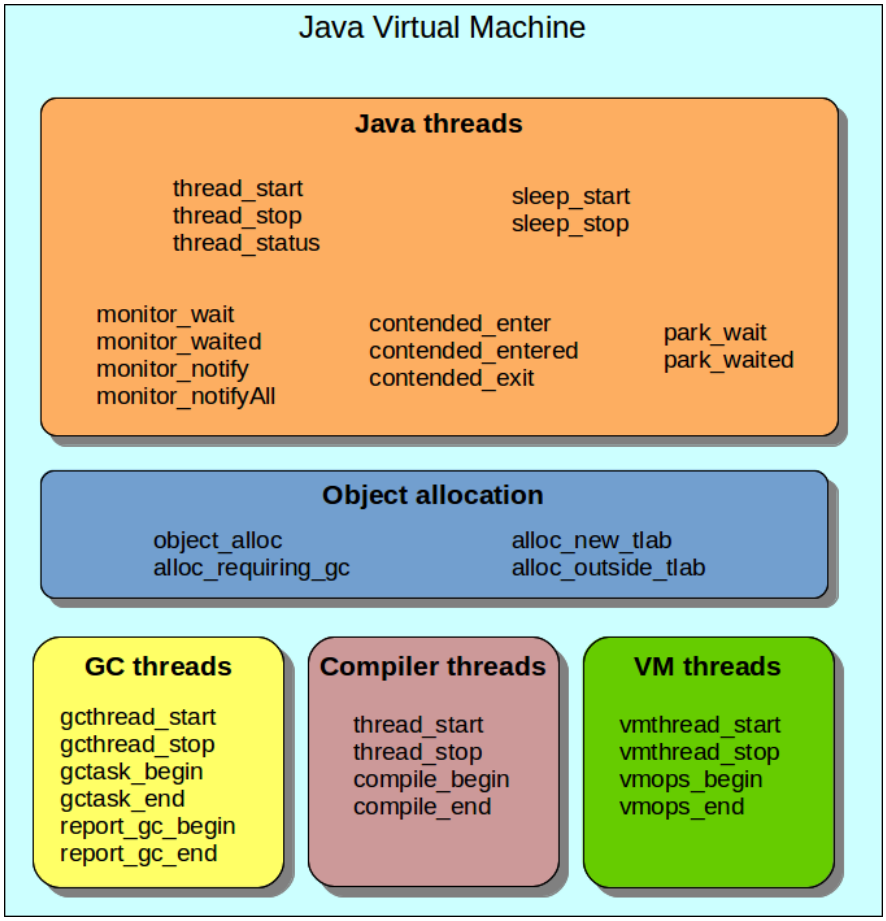


Figure 7.3 Userspace Tracepoints

— VM threads tracepoints (Table 7.5)
— Object allocation tracepoints (Table 7.6)

Kernel Tracing

Kernel tracing provides detailed information about the operating system. The extracted data is very important to understand exactly what is happening in the system during the execution of the application. For example, the application can be scheduled out by the OS scheduler, or blocked for an I/O operation. By tracing the operating system, we can have a better picture of what is running on the same computer.

In this paper, we trace the OS scheduler in order to know which thread is running on which CPU core. This information is important to detect CPU contention and preemption between different applications. The *sched_switch* event is called whenever a thread is scheduled in or out of the CPU, and *sched_waking* when a blocked application is ready to be executed again.

Table 7.2 Java threads events

Tracepoint	Definition
thread_start / thread_stop	A Java thread is started.
thread_sleep_start / thread_sleep_stop	The thread is sleeping
monitor_wait / monitor_waited	The thread is waiting for a monitor
monitor_notify / monitor_notifyAll	The current thread notifies other threads that the monitor is released
contended_enter / contended_entered / contended_exit	Those events show when a thread is blocked on a lock and when it enters and exits a critical section. The lock is defined by a name and an address
park_wait / park_waited	The events indicate when a thread is parked and unparked.

Table 7.3 GC threads events

Tracepoint	Definition
gcthread_start / gcthread_stop	A GC thread is started or stopped
gc_begin/gc_end	Those events give when the name and the id of the garbage collection algorithm exectued by the JVM
gctask_begin / gctask_end	These events show the specific task currently processed by the current GC thread

Table 7.4 JIT compiler events

Tracepoint	Definition
thread_start / thread_stop	A compiler thread is started or stopped
compile_begin / compile_end	A method is being compiled by the thread

We also trace the system calls issued by the target application. The *read* and *write* systems calls are used for file and sockets operations. The *futex* system calls are used to track lock contention between the different threads etc.

On a deeper level, we are using the block layer instrumentation to collect data about disk

Table 7.5 VM threads events

Tracepoint	Definition
vmthread_start / vmthread_stop	A VM thread is started or stopped
vmops_begin / vmops_end	The VM operation is called concurrent if it can be executed without blocking Java threads

Table 7.6 Object allocation events

Tracepoint	Definition
object_alloc	A new object is allocated by the current thread
alloc_new_tlab	The object is stored in a newly created TLAB.The object is stored in a newly created TLAB.
alloc_outside_tlab	The object is created in the shared memory region.
alloc_requiring_gc	The last allocation caused the execution of the garbage collector

usage, and the network layer to track the packets sent over the network.

Profiling Agent

In addition to kernel and userspace events, we want our tool to provide profiling data so that we can inspect hot functions. Unlike other tools, we decided not to use the *AsyncGetCallTraces* function, because it is unable to get the call stack of JVM internal threads. Instead, we created a patch that integrates Libunwind [196] into Lttng-UST. When a userspace event is fired, the tracer walks the call stack and includes it in the payload. *Libunwind* uses optimized caching mechanisms to accelerate the stack walking. If a frame has been unwinded before, it is likely to find it in the cache and thus no further processing is required.

Our profiling agent is basically a JVMTI agent that uses a timer to periodically generate a special UST event. The call stack is walked everytime the profiling event is fired, as shown in Figure 7.4.

In addition to profiling, the provided patch is also useful to know which parts of the code are triggering certain behaviors. For instance, by recovering the call stack of object allocation

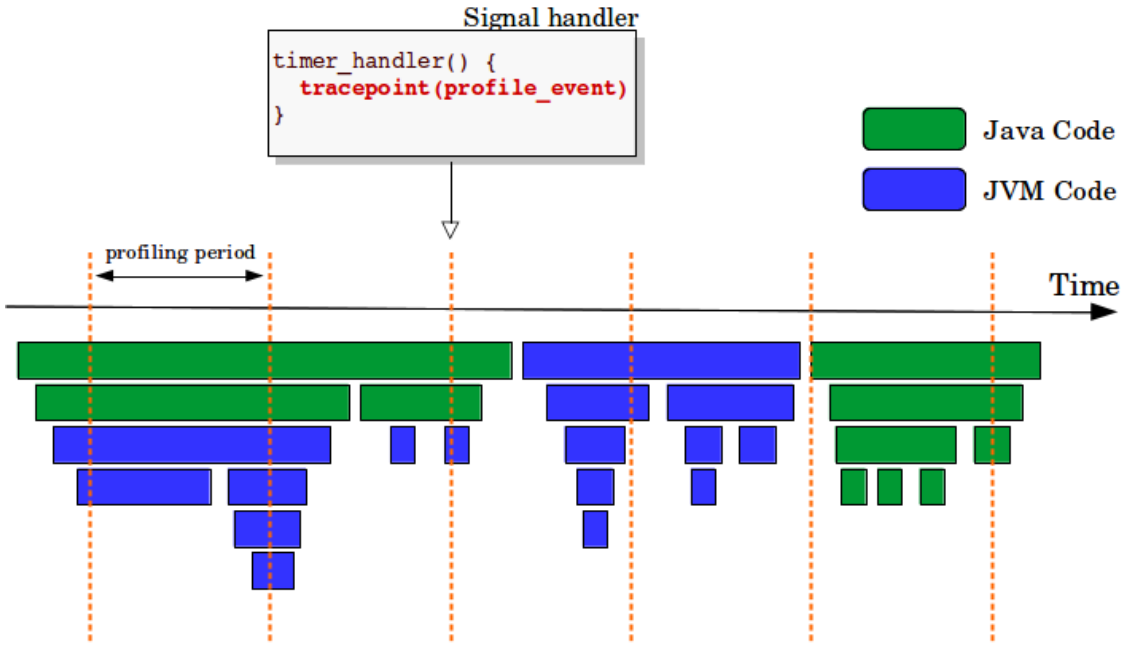


Figure 7.4 Profiling

events, we can detect the functions that are generating most of the allocations.

7.4.2 Trace synchronization

In our analysis, we use Lttng to collect data from different data sources. We are getting information from the Java application, the JVM and from the operating system. We defined three separate channels to collect the events.

— Kernel channel : All kernel events are registered in this channel : system calls, network events, CPU scheduler events, disk events, etc.

— User channel : Userspace events are registered in this channel : thread management, memory allocations, garbage collection etc.

— Profiling channel : Profiling events are written in this channel. The call stack context is activated for this channel.

The synchronization of the different layers is done based on the monotonic clock of the operating system, which insures total ordering. Using timestamps and thread TIDs, we are able to associate events between different layers and define causality relationships between them (Figure 7.5).

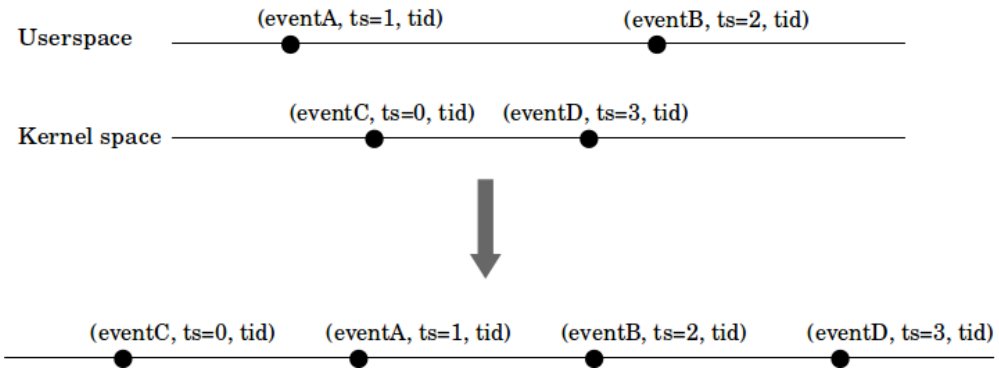


Figure 7.5 Trace synchronization

7.5 Data Analysis and Visualization

7.5.1 Data Model

One of the biggest challenges in tracing is the huge number of events generated, at a very high frequency. Problem detection has to be automated or at least semi-automated. Many approaches have been proposed to simplify data analysis. Naser and al. [139] proposed a technique based on data abstraction to reduce trace size. Related low-level events are grouped into more meaningful higher level compound events. Another approach is to use machine learning techniques to automatically detect anomalies.

In this paper, we use a combination of metric-based abstraction and visual abstraction to present the data in a convenient way. Metric-based abstraction consists of reading the trace events and summarizing them by defining a set of metrics. Just by watching the metrics, the user should be able to have a fairly precise idea about the application performance. Visual abstraction consists in graphically representing the data and showing it on the screen display. We used tracecompass to create views that represent the different components of the system.

Diagnosing a performance issue requires to navigate the trace horizontally, choosing different time ranges, and vertically, zooming in and out on different threads. Instead of reading the trace events every time a new time range is selected, we decided to use a stateful approach that consists in saving the state of the system in an incrementally built database. When the user navigates the trace, the states are queried directly from the database, without the need to reread the trace. Many memory-based data structures are available for time interval based data, such as Segment-tree or R-tree, but using them in our context is not possible due to the huge size of trace files. It is not possible to keep all the necessary information in memory. Traditional relational databases can be used, but previous research has shown that

their query time is not optimal for trace analysis. *The Modeled State System* [155, 156] has proven to be a very efficient data structure for holding the state intervals of the system in a tree-like fashion. It is composed of an *attribute tree* and a *state history tree*. The attribute tree describes the different components of the system, and the state history tree keeps the state of each component throughout time.

We designed an attribute tree that contains all the information necessary to analyze Java applications. A sample of this tree is presented in Figure 7.6. For every thread created by the JVM, we keep two attributes. The user-state attribute describes the state of the thread from the userspace level. A thread can be marked as running, sleeping, waiting for a monitor etc. The kernel-state attribute contains lower-level kernel information about the thread : running on CPU, waiting for futex, executing a system call, etc.

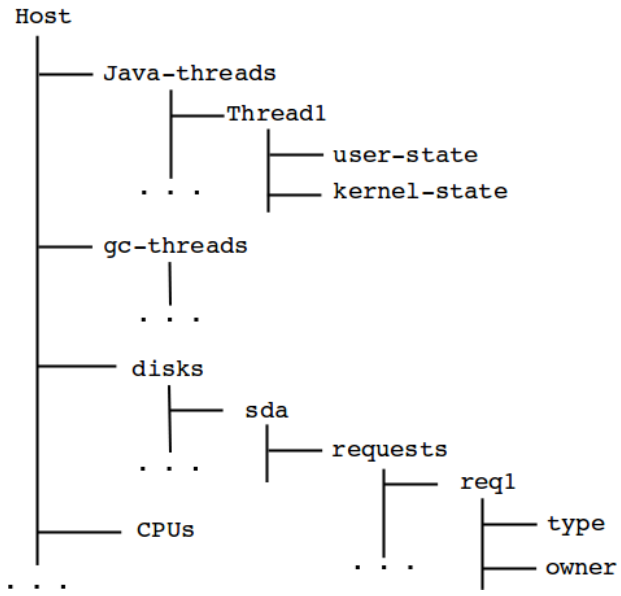


Figure 7.6 Proposed attribute tree

7.5.2 Finite State Machines

In order to efficiently track the state of each system component, we modeled state transitions as finite state machines. In the next subsections, we present the state machines that we defined for each type of thread created by the application.

Java threads Life Cycle

It is important to know the state of each Java thread during execution. Having a clear picture about all the executing threads, and if they are running or not, helps in detecting performance bottlenecks. We inserted many tracepoints into the task manager of the Hotspot JVM in order to get this information. Based on the collected events, we are able to create a model that defines the life cycle of each thread. The different states defined are :

- Running : the thread is executing Java code
- Sleeping : the thread is waiting for a timer
- Blocked on critical section : the thread is waiting to enter a critical section
- Blocked on monitor : the thread is waiting on a monitor. It is waiting for a signal from another thread to be able to execute. It is also possible to define the maximum waiting time before going back to the running state
- Blocked on park : the thread is waiting for a permission to execute. It is also possible to define the maximum waiting time before going back to the running state

The life cycle of Java threads is shown in Figure 7.7

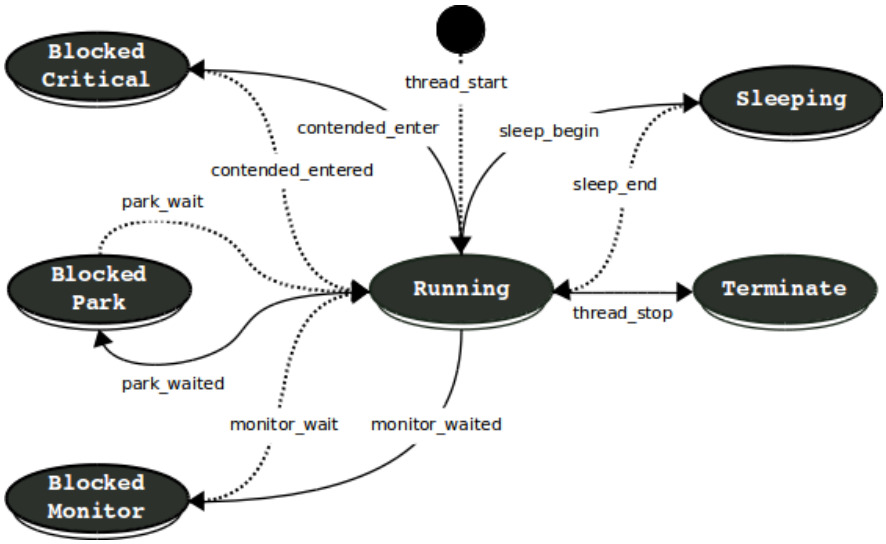


Figure 7.7 Java thread life cycle

GC Threads Life Cycle

GC Thread are created by the JVM to execute garbage collection operations. Instead of collecting the heap with a single thread, the memory space is divided into different regions, and each is collected with a separate thread. Although some critical GC operations cannot be

performed in parallel, using multiple threads greatly increases the speed of garbage collection. The number of GC threads can be manually set using the *-XX :ParallelGCThreads* option.

The events *report_gc_start* and *report_gc_stop* are generated when a GC operation is started and finished. The payload of those events indicates the type of the garbage collector in use. Many GC threads can be used by the JVM to execute garbage collection operations. The events *gctaskthread_start* and *gctaskthread_stop* indicate the lifetime of a GC thread. A GC operation requires different GC tasks. Those tasks can potentially be executed in parallel by different threads. The *gctask_start/stop* events are used to know which task is being executed by which thread.

JIT Threads Life Cycle

Compiler threads are created at the startup of the application and they continue running until the end. A JIT thread can be a C1 or C2 compiler and the number of threads depends on the configuration. JIT threads are basically daemons that stay inactive until a compilation task is required. The tracepoints *compile_begin* and *compile_end* mark the beginning and the end of a compilation task, and their payload contains the name and the class of the compiled function.

VM Threads Life Cycle

VM threads are used to execute internal virtual machine operations. The events *vmthread_start* and *vmthread_stop* indicate when a VM thread is created and killed. VM threads are daemons that are waiting continuously for virtual machine tasks. When a new task appears in the VMOperationQueue, the VM thread executes it and then goes back to sleep. The execution is marked by the *vmops_begin* and *vmops_end* events.

7.5.3 Analysis algorithm

Detecting the root cause of a performance problem requires an advanced analysis that covers all the involved components. In this work, we developed an algorithm to help users detect problems in their Java applications. The algorithm is based on a top-down approach that consists of analyzing higher level data to detect irregularities and then go deeper to provide more precision.

The root cause analysis algorithm is presented in algorithm 2. If the thread spends a lot of time sleeping or blocked, we mark that as a potential problem and we return the related call stacks. If the thread is marked by the JVM as running but it is not executing on a CPU

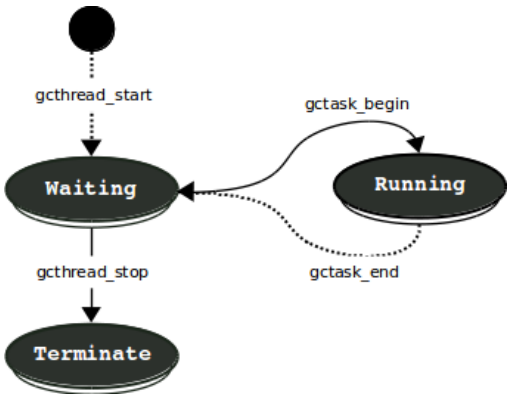


Figure 7.8 Garbage collection thread life cycle

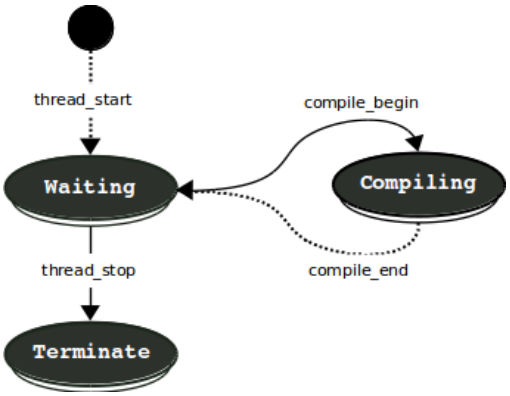


Figure 7.9 Compiler thread life cycle

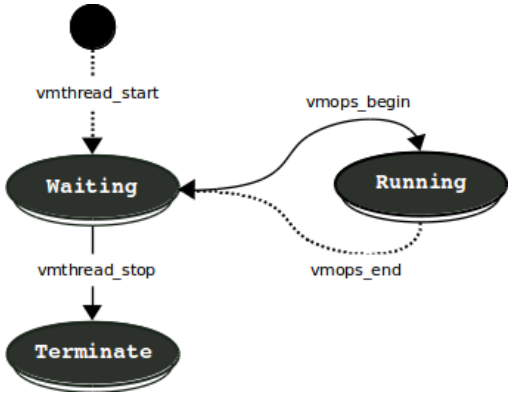


Figure 7.10 VM thread life cycle

core, we investigate problems related to GC, JIT compilation and resource contention. If the thread is running on the CPU but it takes a lot of time to execute, we use the profiler to detect slow functions and we return this information to the user for further analysis.

7.5.4 Visualization

As mentioned above, trace files are generally very big and contain a tremendous amount of information. The trace events are low-level and very difficult to analyze manually. In this paper, we provide a visualization framework that gathers the data from the state system and presents it in a graphical way. Our goal is to have flexible views that offer full visibility of the application, and guide the user in performance troubleshooting. To achieve that, we implemented our views as a part of Trace Compass [123].

Trace Compass is an open source trace visualization tool which offers different views designed

Input: thread
Output: root_causes

```
root_causes ⟵ [ ];
[ running, sleeping, blocked ] ← compute_user_stats(thread);
if sleeping > threshold then
    sleep_callstacks ← getCallstacks(sleep);
    root_causes+ = (sleeping, sleep_callstacks);
end
if blocked > threshold then
    monitor_callstacks ← getCallstacks(monitor);
    park_callstacks ← getCallstacks(park);
    root_causes+ = (monitor, monitor_callstacks);
    root_causes+ = (park, park_callstacks);
end
if the thread is running then
    if GC detected then
        gc_callstacks ← getCallstacks(gc);
        root_causes+ = (gc, gc_callstacks);
    end
    if JIT detected then
        jit_callstacks ← getCallstacks(jit);
        root_causes+ = (jit, jit_callstacks);
    end
    [ on_cpu, syscalls ] ← compute_kernel_stats(thread);
    if syscalls > threshold then
        Get the kernel events involved;
        if Kernel latency (disk/network/cpu contention) then
            root_causes+ = (resource_contention);
        end
    else
        # The thread is running on the cpu;
        profiler_callstacks ← getProfiler();
        root_causes+ = (program_logic, profiler_callstacks);
    end
end
```

Algorithm 2: Analysis algorithm

for specific aspects of the system. Many analysis have been proposed for Trace Compass, such as the critical path analysis [1], virtual machines analysis [197] [198], etc. The data structures and the algorithms proposed in this paper resulted in many additional views for Java applications analysis. The views are synchronized : when the user selects a time range, all the views are updated to show details about that specific region of the trace.

The *Threads view* (figure 7.11) shows the state of all the threads created by the application, including regular Java threads, JIT threads, GC threads and VM threads. This provides an overview of the general behavior of the application and how the different threads interact with each other. The view shows by default the kernel and the userspace states side by side, but the user can still decide to hide unneeded information. Another interesting feature provided by this view is its ability to interact with the *Profiler view*. For example, in Figure 7.11, the user selected a time range to investigate a garbage collection operation, and the profiler view (Figure 7.12) is updated to show the call stacks of the GC thread during the selected range.

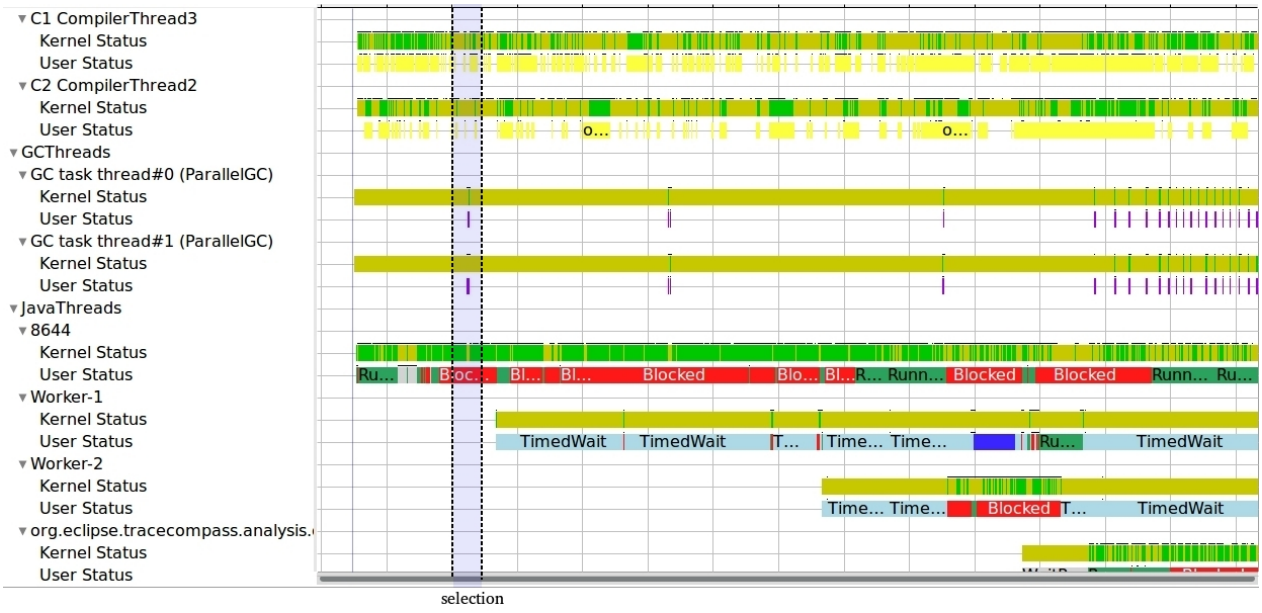


Figure 7.11 Threads view

The *CPU view* (Figure 7.13) shows the threads running on each core of the CPU. This view is very useful to detect preemption. A thread can be seen as running from the userspace perspective but it is not actually executed because it has been preempted by the operating system scheduler, due to CPU contention.

The *lock contention view* shows the list of monitors and the different threads acquiring and releasing them. In Figure 7.14, we see that threads named Thread-n are competing to acquire

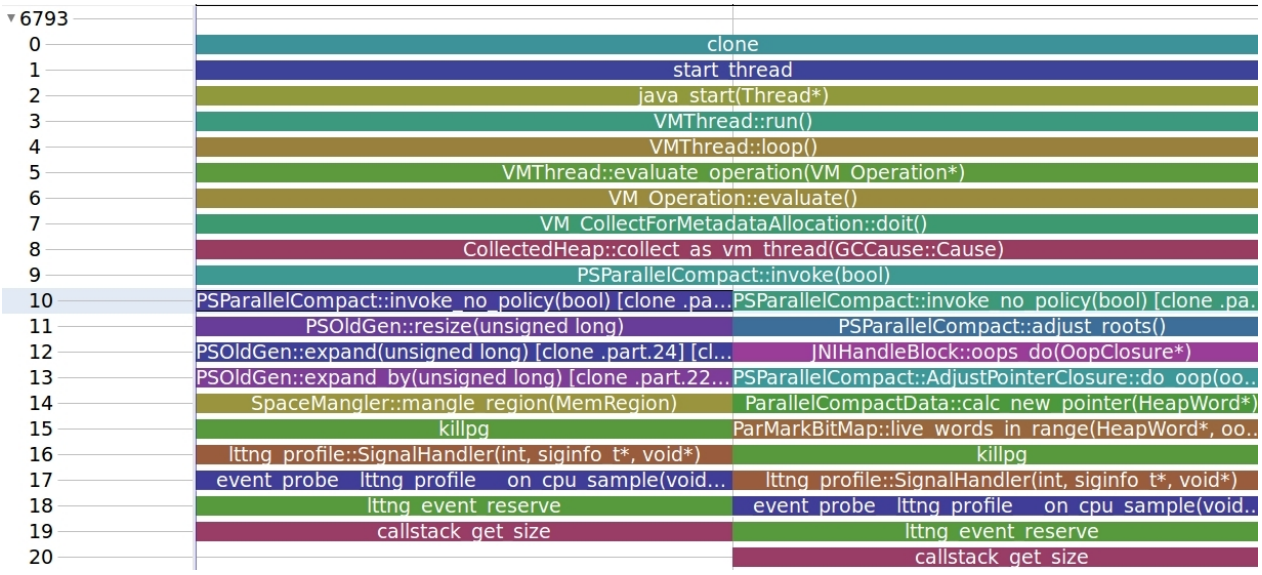


Figure 7.12 Profiler view

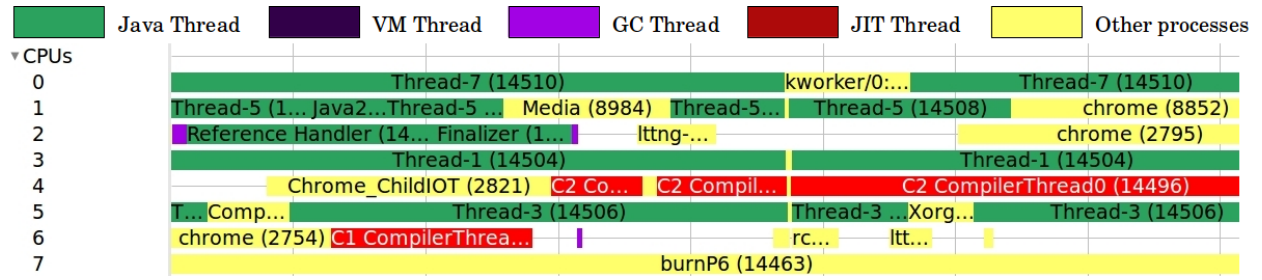


Figure 7.13 CPU view

the monitor BufferedImage.

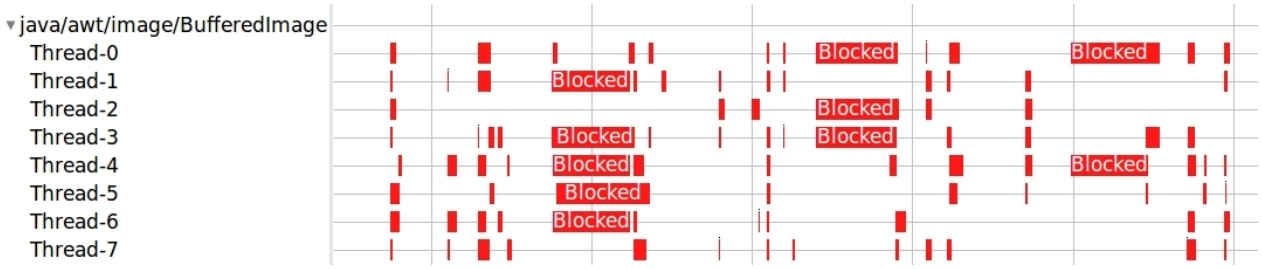


Figure 7.14 Lock contention view

The *memory view* shows the class and the number of objects allocated by the different threads

(Figure 7.15). By recovering the call stack of object allocation events, we can detect which functions are consuming most of the memory.

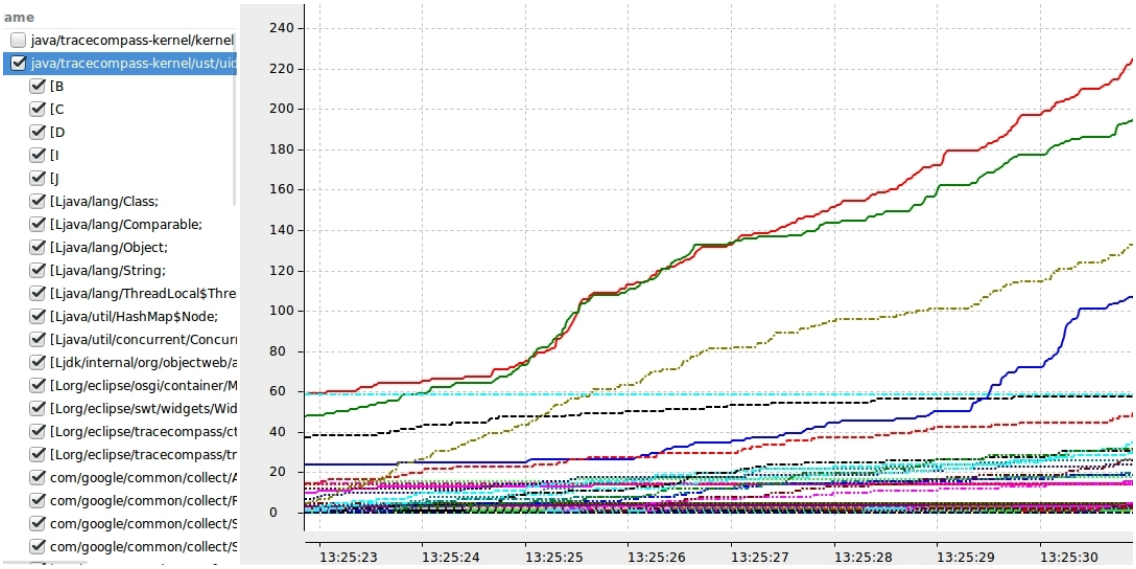


Figure 7.15 Memory usage view

7.6 Use cases

The main contribution of our tool is being able to provide Kernel and userspace information at the same time, providing low-level details about the behavior of Java applications. In this section, we present different use cases where our tool is able to solve challenging problems.

7.6.1 Use Case 1 : CPU contention of processes

Many existing Java analysis tools are able to show the state of the different Java threads at runtime. The problem with their approach is that it is fully based on userspace information. Consequently, the views presented are misleading in a way. When a thread is marked in userspace as running, it does not mean that it is actually being executed on a CPU core. The operating system can decide at any time to schedule out the thread and allocate the CPU core to another application. Unlike other tools, by combining kernel and userspace events, we are able to provide precise information about the CPU cores and which thread is physically running on each core.

The software analyzed in this use case is a photo editing application written in Java. It is implemented as a web service which receives a picture from the network, applies a filter and sends it back over the network. The filter algorithm is fully parallel and can be executed by

multiple threads. Figure 7.16 shows how tasks are distributed : the task size is computed by dividing the total number of pixels by the number of threads. The remainder is then distributed among threads to insure maximum fairness.

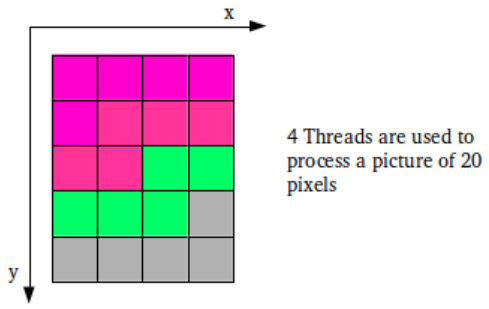


Figure 7.16 Parallel processing of a picture

We conducted some experiments with the goal of evaluating response time of the application. Interestingly, after applying the same filter multiple times on the same picture, we noticed that the execution time is not stable. It fluctuates between 2 and 4 seconds, as shown in Figure 7.17.

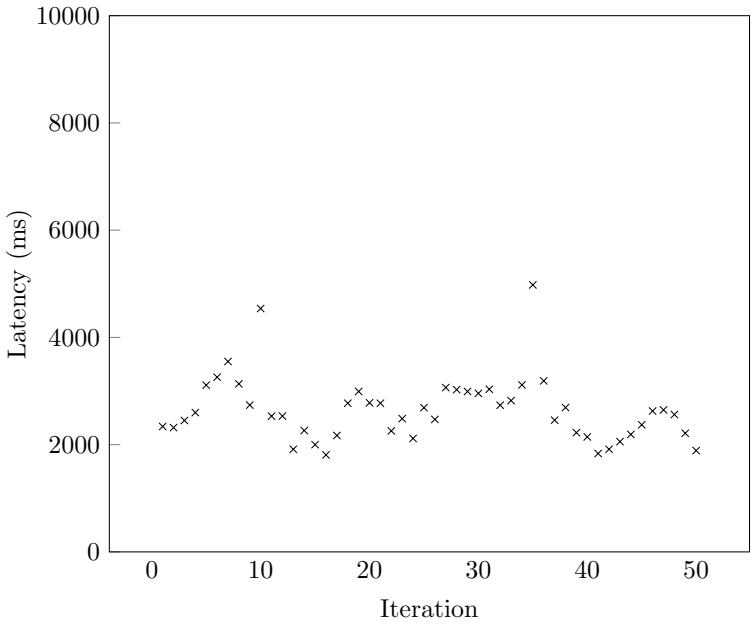


Figure 7.17 Execution time

To dig more, we used our tool to see what is happening exactly in the system during the execution. We activated the userspace tracepoints, as well as the kernel events *sched_switch* and *sched_waiting*, which show when a process is scheduled in and out by the operating

system. Figure 7.18 shows the different processes running on each of the 8 cores in the system. We can see that Java threads are sharing the CPU cores with other system processes colored in yellow. Java threads are sometimes interrupted by other processes. For example, Thread-2 (TID 16683) was interrupted by Chrome (TID 2836) in Figure 7.19.

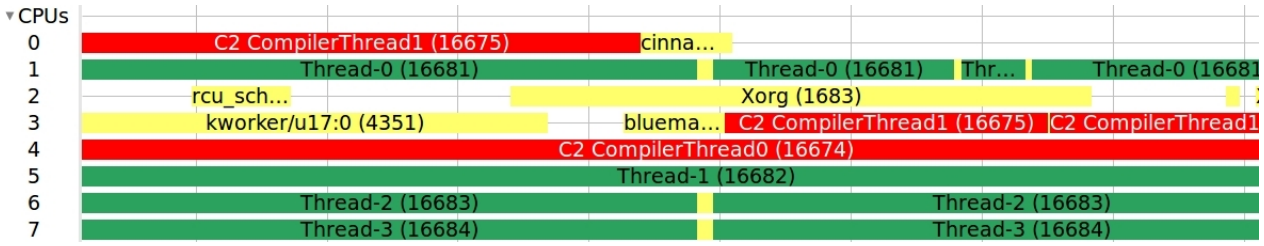


Figure 7.18 Java threads are sharing the CPU cores with other applications

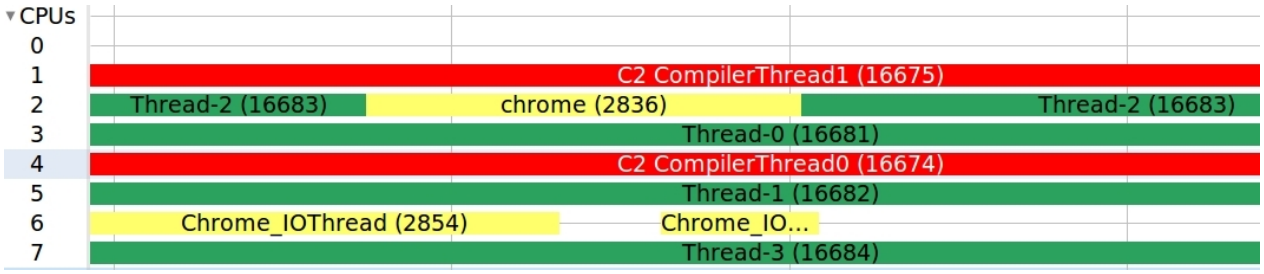


Figure 7.19 A Java thread is interrupted by another application

This execution time variability is not acceptable, since the response time has to be more predictable for the users. The response time should ideally not depend on the other applications running on the same system. To solve this problem, we decided to use the CPU isolation techniques provided by modern systems. We created two *cgroups* : the first contains the Java threads and the other contains the rest of the system processes. Using the *cpuset* command, we associated 4 exclusive cores to the Java application so that they are not disturbed by other processes.

We ran the same experiment again with the new configuration and we used our analysis tool to visualize the behavior of the system. We can see in Figure 7.20 that the Java threads are running on cores 0-3, without any interruption by other system processes. The other processes are running on the other CPU cores, which was the desired behavior. Figure 7.21 shows that the execution time is much more stable than before.

Solving low-level performance problems like this is not possible just by looking at the source

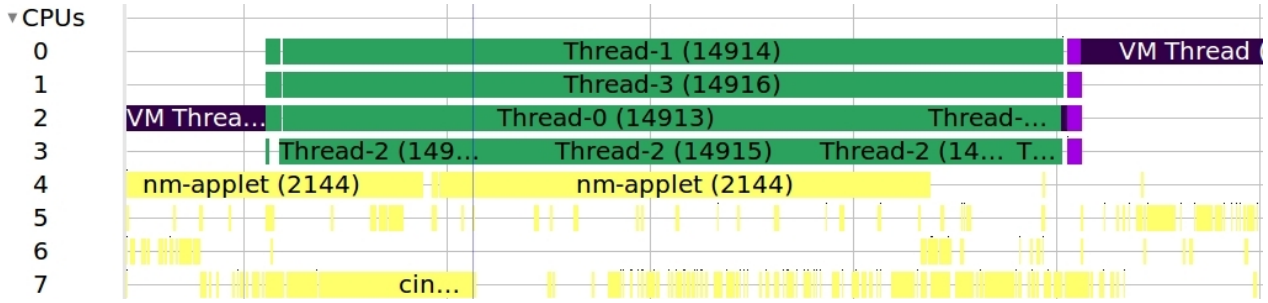


Figure 7.20 Java threads are exclusively running on cores 0,1,2,3

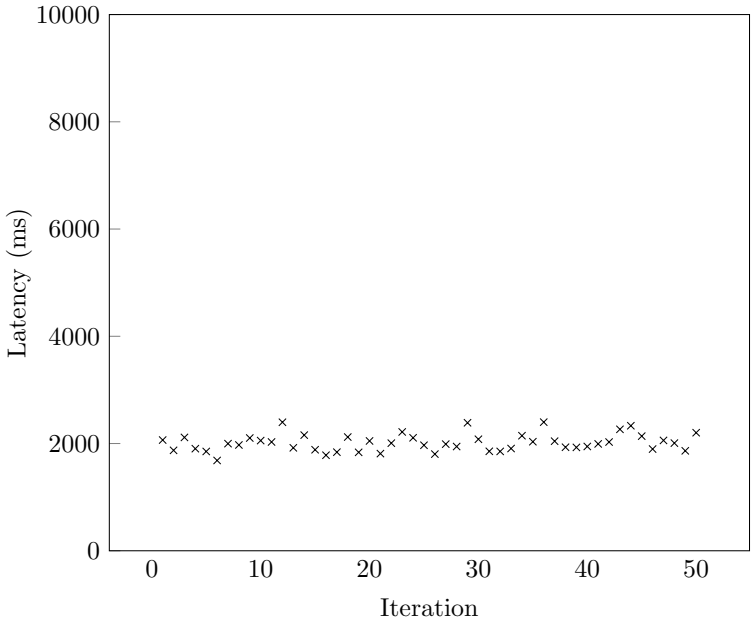


Figure 7.21 Execution time after CPU isolation

code or analyzing JVM events. The only way to see those details is to have full visibility of the system, including operating system internals.

7.6.2 Use Case 2 : File reading latency analysis

Input/Output (I/O) operations are fully managed at the operating system level. For instance, when an application wants to read data from the disk, it uses a read system call to notify the operating system. The OS creates a disk request and registers it in the I/O scheduler waiting queue. The disk scheduler then decides which request has to be issued first to the driver for processing. The different trace events involved in a disk operation are presented in Figure 7.22.

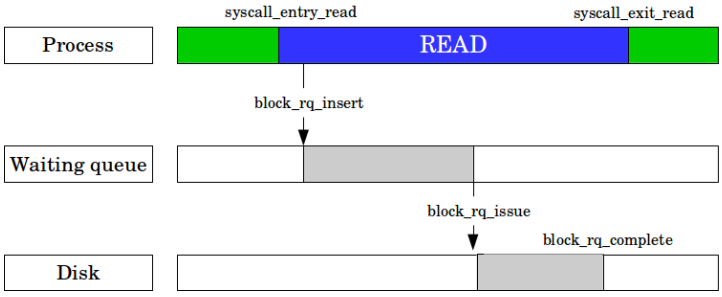


Figure 7.22 Disk request events

In this use case, we analyze a Java application that reads multiple files from the hard disk for later processing. This application is I/O bound and the speed of the disk directly affects the performance of the application. We performed a top-down analysis to understand how the program works and the factors affecting its performance.

We started first with the userspace layer to have a clear picture of the application logic. Figure 7.23 shows that the program is repetitively calling the *read* function which fills the I/O buffers with data from the disk. We notice that the function sometimes takes a longer time to return.

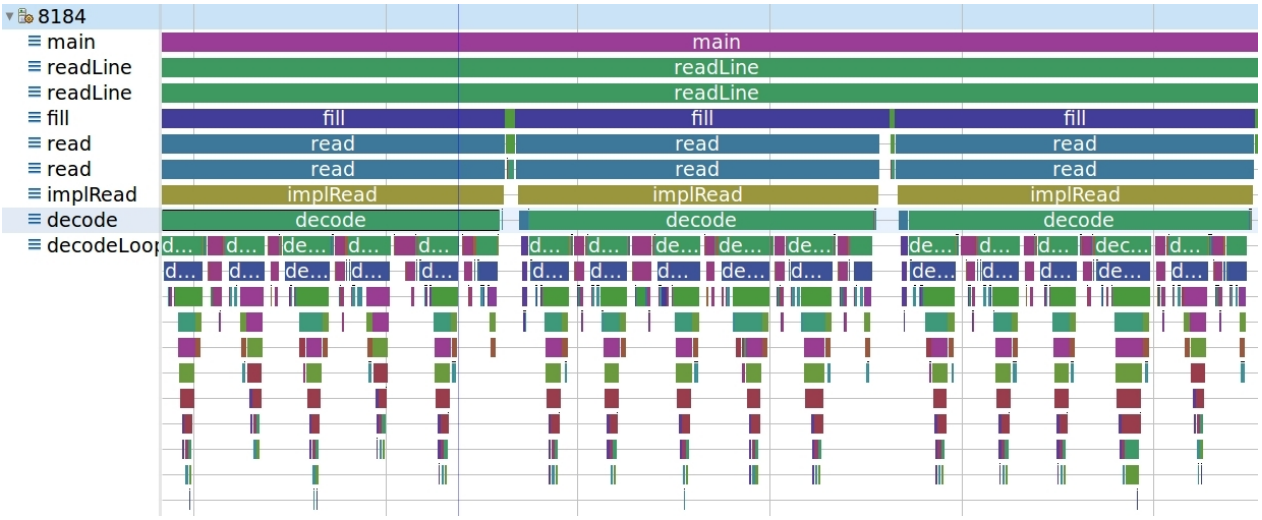


Figure 7.23 Userspace reading

By tracing the system calls (Figure 7.24), we can see that the Java application issues a *read* system call and then is blocked for a certain time until the data becomes available (Figure 7.24).

The waiting time is unstable, even for the same buffer size. In order to understand the

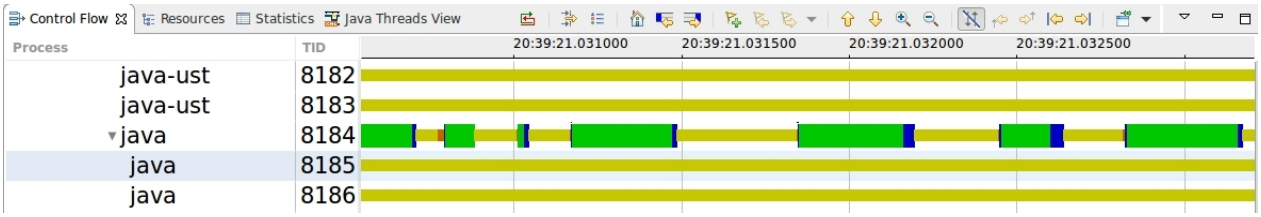


Figure 7.24 Reading system calls

underlying reason, we decided to perform a lower-level analysis that involves the block layer events : *block_rq_insert*, *block_rq_issue* and *block_rq_complete*.

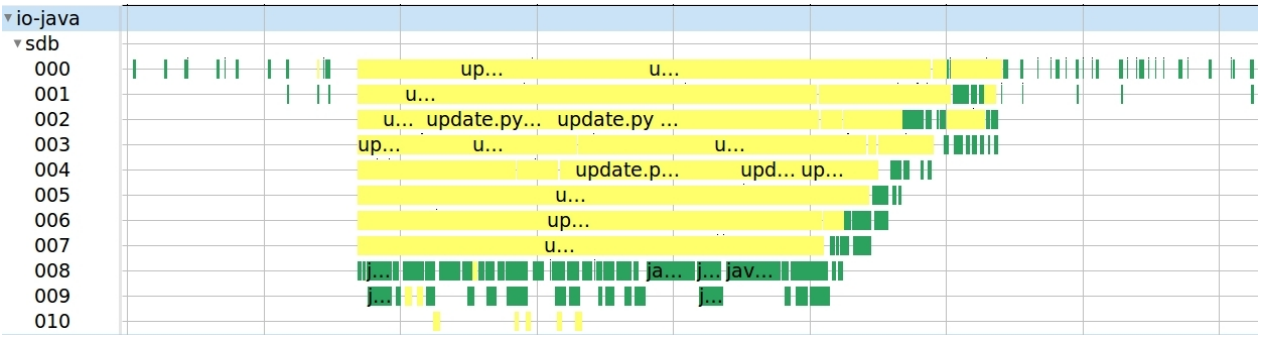


Figure 7.25 Disk usage

Figure 7.25 shows the content of the waiting queue of the hard disk throughout time. The requests colored in green correspond to the target Java application and the yellow ones correspond to other processes. We can see that the Java requests are served very quickly when the waiting queue is empty. However, at some point, a process called *update.py* issued a burst of disk requests, which affected the processing time of Java requests. One of the implications that emerge from these findings is that the performance of an application cannot be evaluated independently. It is very important to use a holistic approach that involves the environment on which it is running.

7.6.3 Use Case 3 : Analysis of runtime parameters

The JVM offers the possibility to select different runtime options like the collection algorithm, the number of JIT threads, etc. It is sometimes difficult to select the options that fit the application and the system environment. In this section we show how, by visualizing the behavior of the application, the programmer can detect and fix misconfigurations.

Garbage collector options

In the first example, we traced a Java application to see if the garbage collector is working properly. Figure 7.26 shows that there are 16 garbage collector threads that are running to perform collection tasks. The blue boxes indicate the name of the GC task executed by each thread. Interestingly, by zooming in, we can see that the threads spend most of their time waiting, they are not all running at the same time (Figure 7.27).

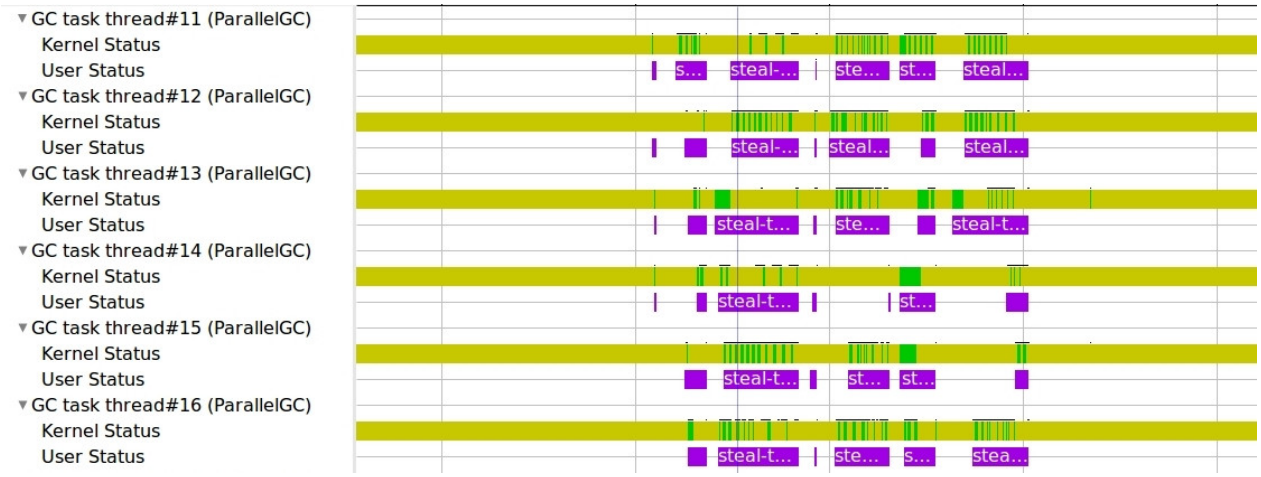


Figure 7.26 16 GC threads are created

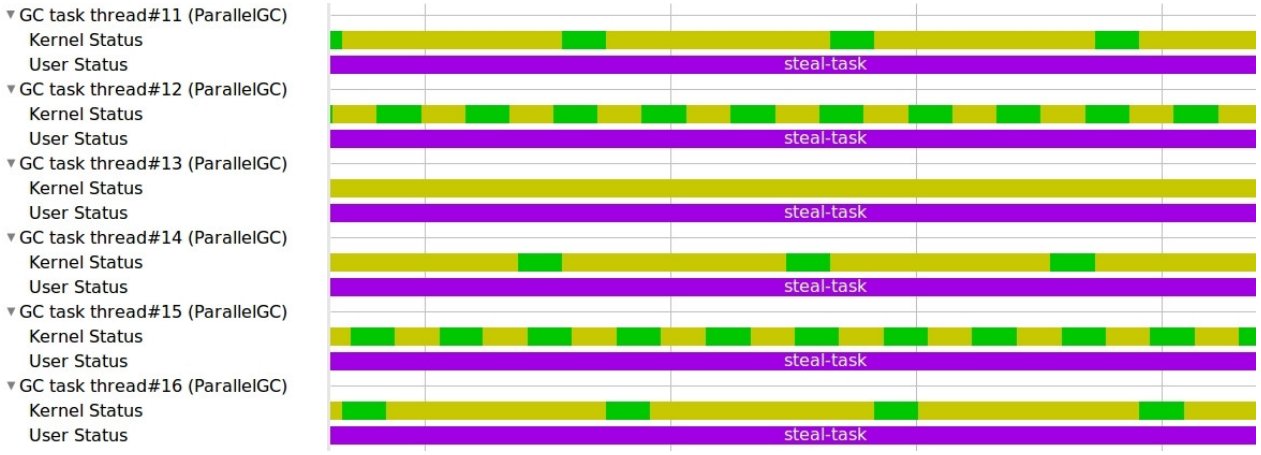


Figure 7.27 The GC threads are not always running at the same time

The reason for the observed behavior is the fact that the system has only 8 CPU cores, and 16 GC threads were created. Those threads cannot all be scheduled at the same time, due

to CPU contention. As shown in Figure 7.28, GC threads are competing with each other for CPU cores.

We can fix this behavior by using the option *-XX :ParallelGCThreads* to set a more appropriate number of parallel GC threads.

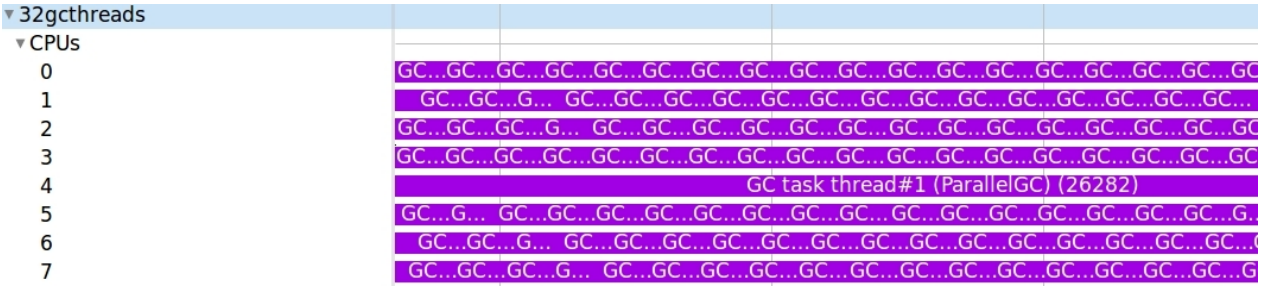


Figure 7.28 There is a CPU contention caused by a big number of GC threads

JIT compiler options

In this subsection, we analyze the startup latency of a Java application. The program takes a very long time to load before running. To detect the problem, we used our tool to observe its behavior from when it is launched until it actually starts running.

Figure 7.29 shows that the JIT compiler is very active before the program starts running. This behavior is surprising since the JIT compiler is only supposed to compile some functions after they were called many times, thus minimally disturbing the main program. By looking at the JVM configuration, we discovered that the program was executed with the *XComp* option, which asks the JIT compiler to compile all functions, even if they are not used frequently.

Removing this options dramatically improved the startup time of the application, because the JIT compiler will only compile the frequently executed functions, as shown in Figure 7.30.

7.7 Evaluation

In this section, we evaluate the overhead introduced by our solution. Our goal is to have minimal impact to insure that tracing does not affect the normal behavior of the monitored application. We used DaCapo 2009 [199], a standard benchmarking suite, to generate different workloads, and we computed for each the tracing overhead and the analysis cost.

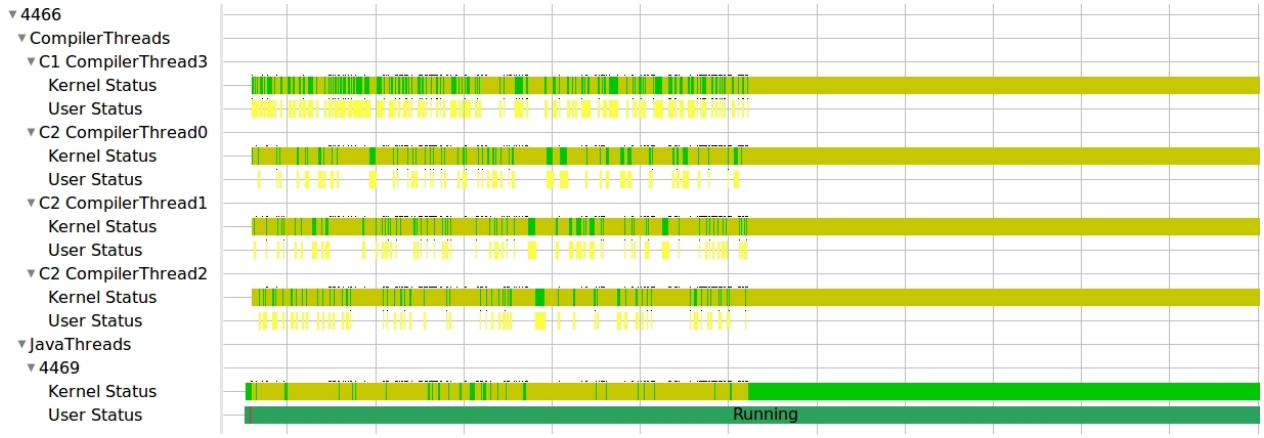


Figure 7.29 Execution with *Xcomp* option

Figure 7.30 Execution with regular JIT tiered compilation

7.7.1 Tracing cost

The tests are executed on an Intel i7-4790 CPU @ 3.60GHz, with 32 GB of main memory and an Intel SSD 530 Series 240 GB hard disk. The system is running Linux Kernel version 4.4 and the traces are gathered using LTTng 2.10.

The experiments are performed with the following configurations :

— *Base :* The tracing is disabled.
— *User* Only userspace tracepoints are enabled
— *Kernel* Only Kernel tracepoints are enabled
— *User+Kernel* Both Kernel and userspace events are enabled.

Our goal is to test our tool with a large range of applications. Dacapo is a Java benchmarking suite containing many real application workloads. We performed the performance evaluation on the following workloads :

— *eclipse :* executes some of the (non-gui) jdt performance tests for the Eclipse IDE.
— *jython :* interprets the pybench Python benchmark.
— *pmd :* analyzes a set of Java classes for a range of source code problems.
— *xalan :* transforms XML documents into HTML.
— *avrora :* simulates a number of programs running on a grid of AVR microcontrollers.

The results of the experiment are presented in Table 7.7, Figure 7.31 and Figure 7.32. The graphs show that tracing does not have a big impact on the execution time of the applications. For the benchmarks *eclipse*, *jython*, *pmd*, and *xalan*, the tracing overhead does not exceed 5.5%, even when kernel and userspace events are enabled at the same time. This low overhead makes our tool acceptable in production systems.

Table 7.7 Tracing overhead

Benchmark	Base (ms)	User (ms)	Kernel (ms)	User+Kernel (ms)	User Overhead (%)	Kernel Overhead (%)	User+Kernel Overhead (%)
eclipse	8039	8252	8343	8486	2.6	3.7	5.5
jython	2543	2651	2639	2652	4.2	3.7	4.2
pmd	1597	1613	1605	1634	1.0	0.1	2.8
xalan	4824	4887	4826	4902	1.3	0.0	1.6
avrora	2512	3244	2616	3380	29.1	4.1	34.5

The only exception happened with the *avrora* workload, where the tracing overhead reached 34.5%. We decided to use our tool to analyze the characteristics of the workload and see the reason for this surprisingly high overhead. Figures 7.33 and 7.34 show the thread states

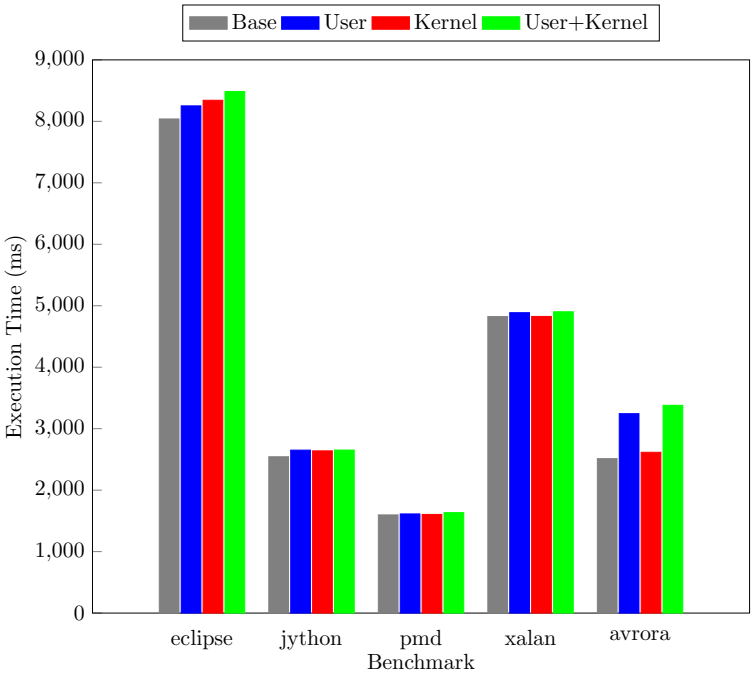


Figure 7.31 Impact of tracing on execution time

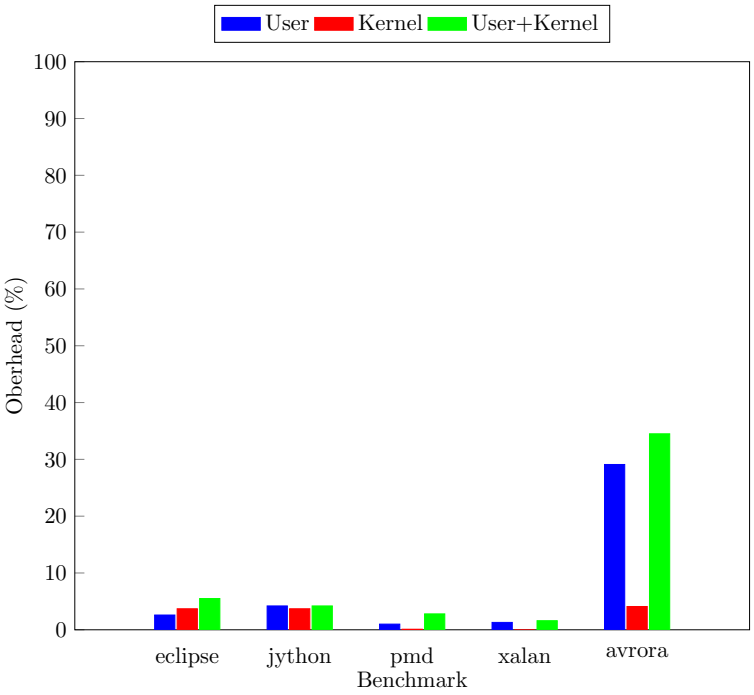


Figure 7.32 Tracing overhead

and the event statistics of *avrora*. We can see that many threads (node-0, node-1, etc) are created. The states of those threads are changing at a very high frequency. In fact, those

threads are competing to enter critical sections protected by synchronization mechanisms. The event statistics view shows that there are 837300 *sched_switch* events, most of them being caused by monitors and lock contention events. This very high events frequency is the cause of the considerable tracing overhead. We can consider this kind of workloads as an atypical worst case scenario.

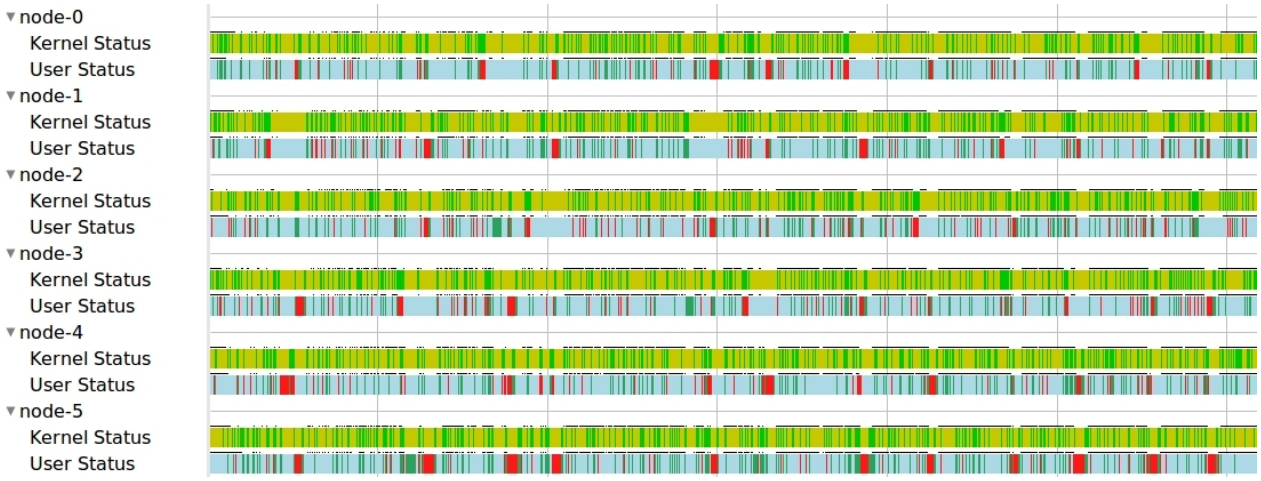


Figure 7.33 *avrora* workload threads view

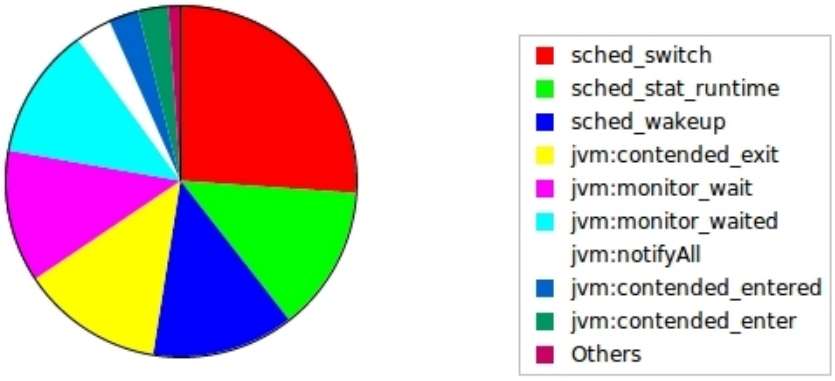


Figure 7.34 *avrora* workload event statistics

7.7.2 Analysis cost

In this paragraph, we present the disk space and the time required to run the analysis. Table 7.8 shows the trace size and the analysis time for the different benchmarks presented previously. Results show that the trace size depends on the frequency of events. The trace

size does not exceed 10 MB for most benchmarks, except *avrora* which generates a trace file of 178MB.

The analysis time is the time required to read the different events in order to generate the data model. Table 7.8 shows that the analysis time is a linear function of the trace size. Bigger traces require more time to read and analyze.

Table 7.8 Analysis cost

Benchmark	Trace size (MB)	Analysis time (ms)
eclipse	7.9	4276
jython	6.4	3348
pmd	9.2	4804
xalan	9.7	5297
avrora	178	93492

7.8 Conclusion

The Java virtual machine is a very powerful, yet complex application. The internal behavior of the JVM is not fully controlled by the developer and, as a result, many performance issues may happen without being noticed. Existing analysis tools offer a limited visibility, since they only use data from the userspace level.

In this paper, we proposed a multilevel performance analysis tool for Java applications that covers the whole application stack. Tracing is used to collect data from the operating system and from the different JVM components, including the garbage collector and the JIT compiler. The analysis module synchronizes and correlates the traces collected from different sources, based on timestamps and event matching rules. The generated model is saved in a disk-based data structure that can be efficiently accessed afterward to generate graphical views.

Different use cases are provided to demonstrate that our tool is able solve real problems, which are challenging to diagnose using traditional tools. In addition, we used the *dacapo* benchmarking suite to evaluate the tracing overhead and the analysis cost. The findings showed that the tracing overhead is acceptable.

The proposed framework is subject to one important limitation. The userspace tracepoints required for the analysis are statically linked to the JVM. As a future work, it would be very interesting to investigate the possibility of using dynamic instrumentation techniques to insert tracepoints at runtime.

Acknowledgments

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, Ciena and EfficiOS.

CHAPITRE 8 DISCUSSION GÉNÉRALE

Dans ce chapitre, nous revenons sur les objectifs de recherche pour évaluer dans quelle mesure ils ont été atteints.

Instrumentation du noyau et des applications Dans nos analyses, nous avons eu besoin de l'instrumentation noyau pour récupérer des informations à partir du système d'exploitation. La majorité de l'instrumentation requise a été réalisée en utilisant les points de trace statiques disponibles dans Linux. L'instrumentation supplémentaire, nécessaire pour quelques analyses, a été obtenue à l'aide d'un module chargé dynamiquement. Ce dernier utilise le mécanisme Kprobe pour s'attacher à des endroits précis du noyau et générer les événements nécessaires. L'instrumentation dynamique introduit une pénalité en termes de surcoût, mais elle permet d'éviter à l'utilisateur de modifier et de recompiler le noyau à partir des sources. Le module développé requiert une option du noyau qui autorise la résolution des symboles. Cette option est généralement activée par défaut dans les distributions Linux, mais il n'y a aucune garantie qu'elle soit disponible dans tous les systèmes.

La même méthodologie a été utilisée pour l'instrumentation des applications en espace utilisateur. Lorsque l'instrumentation statique n'est pas disponible, nous avons utilisé le mécanisme de remplacement de librairies pour instrumenter les applications dynamiquement. Une fonction de rebond est utilisée pour générer le point de trace avant de transmettre l'appel à la librairie. Cette technique n'est pas efficace si la logique du programme ne se base pas sur des librairies externes, ou s'il est compilé statiquement avec ces librairies. Des problèmes peuvent aussi être rencontrés si le programme utilise des mécanismes de protection de l'espace mémoire.

Les analyses présentées dans cette thèse peuvent être exécutées sur des systèmes standards, sans avoir à recompiler le noyau ou les applications. La seule exception est l'étude de la machine virtuelle Java qui nécessite une version instrumentée de OpenJDK. Nous fournissons un correctif que les utilisateurs doivent appliquer [1] pour pouvoir exécuter l'analyse.

Surcoût du traçage Pour chacune des analyses proposées dans cette thèse, nous avons réalisé une série de tests pour mesurer le surcoût introduit par le traçage. Les mesures ont été faites en utilisant des outils d'étalonnages et avec des applications réelles. Les résultats montrent que le surcoût de LTTng est petit avec les cas d'utilisation réalisés, ce qui assure

1. https ://github.com/houssemmh

l'efficacité de nos analyses avec des applications en production.

La taille des traces générées fait partie du surcoût du traçage que nous considérons dans notre recherche. Vu la grande fréquence des événements de stockage, les traces générées sont dans certains cas de très grande taille, ce qui rend l'analyse longue et coûteuse. Pour résoudre ce problème, nous avons proposé des techniques pour réduire le nombre d'événements enregistrés par le traceur. Dans le chapitre 6, nous avons proposé un mécanisme d'agrégation du côté noyau, qui permet de réduire la fréquence de génération des événements de mémoire. Les tests de performance ont montré que cette technique permet effectivement de réduire énormément la taille des traces générées.

Nous avons proposé un autre mécanisme dans le chapitre 7 pour résoudre le problème de la taille des traces dans les systèmes de stockage distribué. Au lieu d'enregistrer tous les événements sur le disque, nous utilisons une session de traçage légère qui surveille les événements les plus importants. Dans cette phase, les événements sont enregistrés dans la mémoire et un algorithme de détection d'anomalies est exécuté à la volée. Si une anomalie est détectée, la trace détaillée est capturée et utilisée pour faire l'analyse approfondie. Ce mécanisme permet de tracer le système en permanence, tout en gardant les traces suffisamment petites pour l'analyse.

Analyse des données Le choix de LTTng comme traceur de référence dans notre recherche nous a permis de collecter des informations à partir de plusieurs sources de données simultanément. En plus des événements de stockage, nous avons aussi tracé les événements de l'ordonnanceur de processus. Ces événements nous ont permis de voir le nom et le statut des programmes actifs sur les cœurs du processeur. Les appels système ont aussi été d'une grande utilité dans la plupart de nos analyses. En parallèle avec le traçage noyau, nous avons utilisé le traçage au niveau utilisateur pour récupérer des informations propres aux applications, afin de comprendre sa logique de fonctionnement. L'algorithme de synchronisation de traces a été d'une grande utilité, car il nous a permis d'assembler plusieurs traces et de les traiter en même temps.

L'utilisation d'un format de trace standard nous a permis de collecter les données simultanément à partir de plusieurs sources. La diversité des sources de données a rendu possible les analyses multiniveau qui donnent une vue globale de la performance des systèmes. Par exemple, nous avons présenté dans le Chapitre 7 des cas d'utilisation où on analyse le ramasse-miette, le compilateur JIT, et l'ordonnanceur des processus et du disque, tous en même temps. Le modèle de l'état (Modeled State System) a aussi été d'une grande utilité, car il peut contenir simultanément l'état des différents composants du système, ce qui est

nécessaire pour effectuer des analyses multiniveau.

La possibilité de relier les points de trace au code source des applications est nécessaire à certaines analyses. Nous avons contribué au développement d'un correctif qui permet de récupérer les piles d'appels des événements de trace en mode utilisateur. Nous avons intégré la librairie *libunwind* avec le traceur LTTng pour traverser la pile lorsqu'un événement est émis. Cette librairie est capable de récupérer la pile d'appels, même si le pointeur de la pile est omis par le compilateur. Cette fonctionnalité permet de voir les emplacements dans le code source qui mènent à certains comportements, ce qui aide le développeur à faire des décisions d'amélioration ou de réusinage du code. Elle peut aussi être utilisée pour effectuer un échantillonnage. Il suffit de charger un module dynamique qui génère un signal périodique et d'associer un gestionnaire à ce signal. Chaque fois que le signal est reçu, un événement de trace est généré et la pile d'appels est retournée dans sa charge utile.

Abstraction des données Il a été possible de représenter les données de la trace d'une manière simple et compréhensible par les utilisateurs. Nous avons pu extraire les métriques populaires à partir des événements de la trace. Une métrique peut être affichée sous forme de courbe ou de tableau, selon les préférences de l'utilisateur.

Des vues graphiques ont été implémentées pour représenter les composants du système. Nous avons modélisé les files d'attente du disque, les fils d'exécution de Java, les démons de Ceph, etc. Cette abstraction graphique aide les utilisateurs à mieux comprendre le fonctionnement du système et de visualiser les comportements étranges qui peuvent se produire. L'utilisation de Trace Compass comme visualiseur de trace a été d'une grande utilité. Les résultats sont affichés dans une interface graphique interactive qui permet aux utilisateurs de facilement naviguer dans la trace.

En résumé, les objectifs de la recherche ont été largement atteints. La technique proposée permet d'étudier efficacement les performances de stockage.

CHAPITRE 9 CONCLUSION

Cette recherche a permis de faire avancer l'état de la connaissance dans le domaine de l'analyse de performance de stockage. Nos contributions incluent :

- La conception et l'implémentation de l'instrumentation nécessaire à l'analyse.
- Des nouveaux mécanismes permettant de réduire la taille des traces générées sans affecter la précision des analyses.
- De nouveaux algorithmes efficaces permettant d'étudier plusieurs aspects de la performance de stockage.
- Des méthodes d'abstraction originales qui permettent de faciliter la compréhension de la trace.

En conclusion, l'approche proposée permet d'étudier les performances de stockage en se basant sur une trace d'exécution. Nous utilisons simultanément le traçage en mode noyau et utilisateur pour collecter des informations sur les différents composants impliqués dans les opérations de stockage. Vu la haute fréquence de quelques événements, nous avons proposé des mécanismes permettant de diminuer le débit de génération d'événements, sans affecter la précision des analyses. Les tests ont montré que le surcoût du traçage est généralement petit et n'altère pas les caractéristiques temporelles des programmes observés.

Une approche à états a été utilisée pour optimiser le temps d'exécution des analyses. Lors de la première lecture de la trace, on génère une base de données qui contient l'historique des états des composants du système. Le but de cette étape est d'éviter la relecture des événements de trace lors de l'exécution des analyses.

Les analyses proposées dans notre recherche incluent l'étude de la performance du stockage de masse, du stockage distribué et du stockage en mémoire. Toutes ces analyses ont été intégrées dans le même outil et peuvent être exécutées simultanément. Ceci permet à l'utilisateur d'avoir une vue complète sur la performance du système observé dans la même interface graphique.

Les cas d'utilisation présentés montrent l'utilité de notre approche dans le débogage des performances de stockage. Nos analyses ont permis de détecter des problèmes de performance difficilement identifiables et d'aider les utilisateurs à améliorer les performances de leurs systèmes.

9.1 Limitations de la solution proposée

Une partie des analyses proposées se basent l'instrumentation statique des applications. Il est nécessaire que les points de trace soient activés au moment de la compilation pour avoir accès aux informations requises. Ceci n'est pas possible si le code source est non disponible ou si l'application cible est déjà en cours d'exécution. Dans ce cas, des techniques d'instrumentation dynamique doivent être considérées.

Les analyses que nous avons proposées permettent d'avoir une bonne visibilité sur la performance du système observé. Par contre, les règles de détection des anomalies utilisées se basent principalement sur les latences des requêtes. En d'autres termes, une requête est considérée comme problématique si sa latence dépasse un certain seuil défini par l'utilisateur. Cette définition d'anomalie est simpliste et ne permet pas de détecter toutes les classes de problèmes. Des algorithmes de détection plus intelligents pourraient être proposés.

9.2 Améliorations futures

Les analyses proposées dans notre recherche s'exécutent hors ligne. Autrement, les traces doivent être d'abord enregistrées sur le disque avant d'être analysées. Les nouvelles versions de LTTng supportent les sessions de traçage en direct. La possibilité d'exécuter les analyses en continu, au moment du traçage, est une piste intéressante à étudier.

Une autre amélioration pertinente consisterait à utiliser des algorithmes d'apprentissage machine pour détecter et classifier les problèmes de performance. Vu la diversité des problèmes liés à la performance de stockage, il serait intéressant d'avoir des mécanismes de détection d'anomalies qui ne comptent pas seulement sur l'expertise des utilisateurs. En combinant ces mécanismes de détection avec la possibilité d'exécuter les analyses en continu, on pourrait mettre en place un système intelligent qui génère des alertes dès qu'un problème de performance sévère se produit.

RÉFÉRENCES

[1] F. Giraldeau et M. Dagenais, "Wait analysis of distributed systems using kernel tracing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, nº. 8, p. 2450–2461, 2016.

[2] V. Tarasov *et al.*, "Benchmarking file system benchmarking : It* is* rocket science." dans *In Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS XIII), Napa, California. Berkeley, CA*, May 9-11, 2011.

[3] M. Desnoyers et M. R. Dagenais, "The lttng tracer : A low impact performance and behavior monitor for gnu/linux," dans *OLS (Ottawa Linux Symposium)*, vol. 2006. Citeseer, 2006, p. 209–224.

[4] S. R. Kleiman *et al.*, "Vnodes : An architecture for multiple file system types in sun unix." dans *USENIX Summer*, vol. 86, 1986, p. 238–247.

[5] R. Love, *Linux Kernel Development*, 3e éd. Addison-Wesley Professional, 2010, ISBN : 9780672329463 0672329468.

[6] S. Pratt et D. A. Heger, "Workload dependent performance evaluation of the linux 2.6 i/o schedulers," dans *Proceedings of the Linux symposium*, vol. 2, 2004, p. 425–448.

[7] Deadline scheduler documentation. [03 Feb 2019]. [En ligne]. Disponible : https://www.kernel.org/doc/Documentation/block/deadline-iosched.txt

[8] S. Iyer et P. Druschel, "Anticipatory scheduling : A disk scheduling framework to overcome deceptive idleness in synchronous i/o," *SIGOPS Oper. Syst. Rev.*, vol. 35, nº. 5, p. 117–130, oct. 2001, doi : 10.1145/502059.502046. [En ligne]. Disponible : http://doi.acm.org/10.1145/502059.502046

[9] Complete fairness queueing (cfq) scheduler documentation. [03 Feb 2019]. [En ligne]. Disponible : https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt

[10] M. Budiu, "A dual-disk file system : ext4," 16 Apr 1997.

[11] O. Rodeh, J. Bacik et C. Mason, "Btrfs : The linux b-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, nº. 3, p. 9, 2013.

[12] B. Pawlowski *et al.*, "The nfs version 4 protocol," dans *In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000*, 2000.

[13] J. H. Howard, "An overview of the andrew file system," dans *in Winter 1988 USENIX Conference Proceedings*, 1988, p. 23–26.

[14] Iometer : I/o subsystem measurement and characterization tool. [18 Jan 2019]. [En ligne]. Disponible : http://www.iometer.org

[15] Iozone filesystem benchmark. [18 Jan 2019]. [En ligne]. Disponible : http://www.iozone.org/

[16] Bonnie++. [18 Jan 2019]. [En ligne]. Disponible : https://www.coker.com.au/bonnie++/

[17] C. Lee *et al.*, "F2fs : A new file system for flash storage." dans *FAST*, Santa Clara, CA, 2015, p. 273–286.

[18] Fio : Flexible i/o tester. [18 Jan 2019]. [En ligne]. Disponible : https://github.com/axboe/fio

[19] R. McDougall et J. Mauro, "Filebench," *URL : http ://www. nfsv4bat. org/Documents/-nasconf/2004/filebench. pdf (Cited on page 56.)*, 2005.

[20] A. Traeger *et al.*, "A nine year study of file system and storage benchmarking," *ACM Transactions on Storage (TOS)*, vol. 4, n°. 2, p. 5, 2008.

[21] J. K. Ousterhout *et al.*, *A trace-driven analysis of the UNIX 4.2 BSD file system.* ACM, 1985, vol. 19, n°. 5.

[22] R. Floyd, "Short-term file reference patterns in a unix environment," 01 1986.

[23] M. G. Baker *et al.*, "Measurements of a distributed file system," dans *ACM SIGOPS Operating Systems Review*, vol. 25, n°. 5. ACM, 1991, p. 198–212.

[24] A. J. Smith, "Disk cache—miss ratio analysis and design considerations," *ACM Transactions on Computer Systems (TOCS)*, vol. 3, n°. 3, p. 161–203, 1985.

[25] C. Ruemmler et J. Wilkes, *UNIX disk access patterns.* Hewlett-Packard Laboratories, 1992.

[26] R. A. Floyd et C. S. Ellis, "Directory reference patterns in hierarchical file systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, n°. 2, p. 238–247, 1989.

[27] J. M. Porcar, "File migration in distributed computer systems," Lawrence Berkeley Lab., CA (USA), Rapport technique, 1982.

[28] O. Kure, "Optimization of file migration in distributed systems," 1988.

[29] R. Stata, "File systems with multiple file implementations," Mémoire de maîtrise, Massachusetts Institute of Technology, 1992.

[30] C. Staelin et H. Garcia-Molina, "Smart filesystems." dans *USENIX Winter*, vol. 91, 1991, p. 45–52.

[31] S. Strange, “Analysis of long-term unix file access patterns for application to automatic file migration strategies,” *Technical report CSD-92-700*, 1992.

[32] S. D. Gribble *et al.*, “Self-similarity in file systems,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, n°. 1, p. 141–150, 1998.

[33] N. Agrawal, A. C. Arpaci-Dusseau et R. H. Arpaci-Dusseau, “Towards realistic file-system benchmarks with codemri,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, n°. 2, p. 52–57, 2008.

[34] Z. Ren *et al.*, “igen : A realistic request generator for cloud file systems benchmarking,” dans *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, 2016, p. 343–350.

[35] Z. Weiss *et al.*, “Root : Replaying multithreaded traces with resource-oriented ordering,” dans *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, p. 373–387, doi : 10.1145/2517349.2522734.

[36] N. Zhu *et al.*, “Tbbt : scalable and accurate trace replay for file server evaluation,” dans *USENIX Conference on File and Storage Technologies*, vol. 5, 2005, p. 24–24.

[37] T. E. Pereira, F. Brasileiro et L. Sampaio, “File system trace replay methods through the lens of metrology,” dans *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2016, p. 1–15.

[38] A. Haghdoost *et al.*, “hfplayer : Scalable replay for intensive block i/o workloads,” *ACM Transactions on Storage (TOS)*, vol. 13, n°. 4, p. 39, 2017.

[39] C. Mason. Seekwatcher. [18 September 2016]. [En ligne]. Disponible : http://oss.oracle.com/~mason/seekwatcher

[40] B. Donie. Ioprof. [03 Feb 2019]. [En ligne]. Disponible : https://github.com/01org/ioprof

[41] A. Brunelle. Btt. [03 Feb 2019]. [En ligne]. Disponible : http://www.cse.unsw.edu.au/~aaronc/iosched/doc/btt.html

[42] Oracle zfs storage appliance administration guide. [18 Jan 2019]. [En ligne]. Disponible : https://docs.oracle.com/cd/E78901_01/pdf/E78912.pdf

[43] B. Gregg, “Visualizing system latency,” *Communications of the ACM*, vol. 53, n°. 7, p. 48–54, 2010.

[44] S. Goldshtein, D. Zurbalev et I. Flatow, *Performance Measurement*, 01 2012, p. 7–59.

[45] O. Rodeh, H. Helman et D. Chambliss, “Visualizing block io workloads,” *ACM Transactions on Storage (TOS)*, vol. 11, n°. 2, p. 6, 2015.

[46] R. Sandberg *et al.*, "Design and implementation of the sun network filesystem," dans *Proceedings of the Summer USENIX conference*, Norwood, MA, USA, 1985, p. 119–130.

[47] J. Kubiatowicz *et al.*, "Oceanstore : An architecture for global-scale persistent storage," dans *ACM SIGARCH Computer Architecture News*, vol. 28, n°. 5. ACM, 2000, p. 190–201.

[48] A. Adya *et al.*, "Farsite : Federated, available, and reliable storage for an incompletely trusted environment," *ACM SIGOPS Operating Systems Review*, vol. 36, n°. SI, p. 1–14, 2002.

[49] D. Dolev et H. R. Strong, "Authenticated algorithms for byzantine agreement," *SIAM Journal on Computing*, vol. 12, n°. 4, p. 656–666, 1983.

[50] R. B. Ross, R. Thakur *et al.*, "Pvfs : A parallel file system for linux clusters," dans *Proceedings of the 4th annual Linux showcase and conference*, 2000, p. 391–430.

[51] P. F. Corbett *et al.*, "Parallel access to files in the vesta file system," dans *Supercomputing'93. Proceedings.* IEEE, 1993, p. 472–481.

[52] F. B. Schmuck et R. L. Haskin, "Gpfs : A shared-disk file system for large computing clusters." dans *USENIX Conference on File and Storage Technologies*, vol. 2, n°. 19, Berkeley, CA, USA, 2002.

[53] O. Rodeh et A. Teperman, "zfs-a scalable distributed file system using object disks," dans *Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on.* IEEE, 2003, p. 207–218.

[54] P. J. Braam *et al.*, "The lustre storage architecture," 2004.

[55] M. Mesnier, G. R. Ganger et E. Riedel, "Object-based storage," *IEEE Communications Magazine*, vol. 41, n°. 8, p. 84–90, 2003.

[56] H. Tang *et al.*, "A self-organizing storage cluster for parallel data-intensive applications," dans *Proceedings of the 2004 ACM/IEEE conference on Supercomputing.* IEEE Computer Society, 2004, p. 52.

[57] S. A. Weil *et al.*, "Ceph : A scalable, high-performance distributed file system," dans *Proceedings of the 7th symposium on Operating systems design and implementation.* USENIX Association, 2006, p. 307–320.

[58] Glusterfs documentation. [03 Feb 2019]. [En ligne]. Disponible : https://docs.gluster.org/en/latest/

[59] S. A. Weil *et al.*, "Rados : a scalable, reliable storage service for petabyte-scale storage clusters," dans *Proceedings of the 2nd international workshop on Petascale data storage : held in conjunction with Supercomputing'07.* ACM, 2007, p. 35–44.

[60] C. Systems. Introduction to cisco ios netflow. [03 Feb 2019]. [En ligne]. Disponible : http://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/prod_white_paper0900aecd80406232.html

[61] W. Stallings, *SNMP,SNMPV2,Snmpv3,and RMON 1 and 2*, 3e éd. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1998.

[62] A. Hussain *et al.*, "Experiences with a continuous network tracing infrastructure," dans *Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*. ACM, 2005, p. 185–190.

[63] R. Fonseca *et al.*, "X-trace : A pervasive network tracing framework," dans *Proceedings of the 4th USENIX conference on Networked systems design & implementation*. USENIX Association, 2007, p. 20–20.

[64] P. Barham *et al.*, "Using magpie for request extraction and workload modelling," dans *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA : USENIX Association, 2004, p. 18–18. [En ligne]. Disponible : http://dl.acm.org/citation.cfm?id=1251254.1251272

[65] B. H. Sigelman *et al.*, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Rapport technique, 2010. [En ligne]. Disponible : https://research.google.com/archive/papers/dapper-2010-1.pdf

[66] F. Wang *et al.*, "Performance and scalability evaluation of the ceph parallel file system," dans *Proceedings of the 8th Parallel Data Storage Workshop*. ACM, 2013, p. 14–19.

[67] D. Van der Ster et A. Wiebalck, "Building an organic block storage service at cern with ceph," dans *Journal of Physics : Conference Series*, vol. 513, no. 4. IOP Publishing, 2014, p. 042047.

[68] G. Donvito, G. Marzulli et D. Diacono, "Testing of several distributed file-systems (hdfs, ceph and glusterfs) for supporting the hep experiments analysis," dans *Journal of physics : Conference series*, vol. 513, no. 4. IOP Publishing, 2014, p. 042014.

[69] B. Depardon, G. Le Mahec et C. Séguin, "Analysis of Six Distributed File Systems," Research Report, févr. 2013, [23 February 2018]. [En ligne]. Disponible : https://hal.inria.fr/hal-00789086

[70] D.-Y. Lee *et al.*, "Understanding write behaviors of storage backends in ceph object store," dans *Proceedings of the 2017 IEEE International Conference on Massive Storage Systems and Technology*, vol. 10, 2017.

[71] X. Zhang *et al.*, “Fsobserver : A performance measurement and monitoring tool for distributed storage systems,” dans *Network and Parallel Computing*, F. Zhang *et al.*, édit. Muroran, Japan : Springer International Publishing, 2018, p. 142–147.

[72] M. Poat et J. Lauret, “Performance and advanced data placement techniques with ceph’s distributed storage system,” dans *Journal of Physics : Conference Series*, vol. 762, nº. 1. IOP Publishing, 2016, p. 012025.

[73] X. Zhang *et al.*, “Fsobserver : A performance measurement and monitoring tool for distributed storage systems,” dans *IFIP International Conference on Network and Parallel Computing*. Springer, 2018, p. 142–147.

[74] Tracing ceph with blkin. [03 Feb 2019]. [En ligne]. Disponible : http://docs.ceph.com/docs/master/dev/blkin/

[75] B. H. Sigelman *et al.*, “Dapper, a large-scale distributed systems tracing infrastructure,” Google, Inc., Rapport technique, 2010, [03 Feb 2019]. [En ligne]. Disponible : https://research.google.com/archive/papers/dapper-2010-1.pdf

[76] End-to-end performance visualization in ceph. [03 Feb 2019]. [En ligne]. Disponible : http://victoraraujo.me/babeltrace-zipkin/

[77] E. Thereska *et al.*, “Stardust : tracking activity in a distributed storage system,” dans *ACM SIGMETRICS Performance Evaluation Review*, vol. 34, nº. 1. ACM, 2006, p. 3–14.

[78] M. Abd-El-Malek *et al.*, “Ursa minor : Versatile cluster-based storage,” dans *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, ser. FAST’05. Berkeley, CA, USA : USENIX Association, 2005, p. 5–5. [En ligne]. Disponible : http://dl.acm.org/citation.cfm?id=1251028.1251033

[79] M. Gorman, *Understanding the Linux Virtual Memory Manager*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 2004, isbn : 0131453483.

[80] M. S. Johnstone et P. R. Wilson, “The memory fragmentation problem : solved ?” dans *ACM SIGPLAN Notices*, vol. 34, nº. 3. ACM, 1998, p. 26–36.

[81] P. R. Wilson *et al.*, “Dynamic storage allocation : A survey and critical review,” dans *Memory Management*. Springer, 1995, p. 1–116.

[82] C. Del Rosso, “Reducing internal fragmentation in segregated free lists using genetic algorithms,” dans *Proceedings of the 2006 international workshop on Workshop on interdisciplinary software engineering research*. ACM, 2006, p. 57–60.

[83] K. C. Knowlton, “A fast storage allocator,” *Communications of the ACM*, vol. 8, nº. 10, p. 623–624, 1965.

[84] J. L. Peterson et T. A. Norman, “Buddy systems,” *Communications of the ACM*, vol. 20, n°. 6, p. 421–431, 1977.

[85] E. D. Berger *et al.*, “Hoard : A scalable memory allocator for multithreaded applications,” *ACM Sigplan Notices*, vol. 35, n°. 11, p. 117–128, 2000.

[86] W. Gloger. Ptmalloc. [03 Feb 2019]. [En ligne]. Disponible : http://www.malloc.de/en/

[87] S. Ghemawat et P. Menage. Tcmalloc : Thread-caching malloc. [03 Feb 2019]. [En ligne]. Disponible : http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[88] J. Evans, “A scalable concurrent malloc (3) implementation for freebsd,” dans *Proc. of the BSDCan Conference, Ottawa, Canada*, 2006.

[89] C. Lameter. Slab allocators in the linux kernel. [03 Feb 2019]. [En ligne]. Disponible : http://events.linuxfoundation.org/sites/events/files/slides/slaballocators.pdf

[90] Slub : The unqueued slab allocator. [03 Feb 2019]. [En ligne]. Disponible : http://lwn.net/Articles/229096/

[91] R. E. Griswold et G. M. Townsend, *The visualization of dynamic memory management in the icon programming language*. University of Arizona, Department of Computer Science, 1989.

[92] T. Printezis et R. Jones, *GCspy : an adaptable heap visualisation framework*. ACM, 2002, vol. 37, n°. 11.

[93] A. M. Cheadle *et al.*, “Visualising dynamic memory allocators,” dans *Proceedings of the 5th international symposium on Memory management*. ACM, 2006, p. 115–125.

[94] S. Moreta et A. Telea, “Visualizing Dynamic Memory Allocations,” dans *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, juin 2007, p. 31–38.

[95] M. Jurenz *et al.*, “Memory allocation tracing with vampirtrace,” dans *Proceedings of the 7th International Conference on Computational Science, Part II*, ser. ICCS ’07. Berlin, Heidelberg : Springer-Verlag, 2007, p. 839–846, doi : 10.1007/978-3-540-72586-2118. [En ligne]. Disponible : http://dx.doi.org/10.1007/978-3-540-72586-2118

[96] S. P. Reiss, “Visualizing the Java heap to detect memory problems,” dans *Visualizing Software for Understanding and Analysis, 2009. VISSOFT 2009. 5th IEEE International Workshop on*. IEEE, 2009, p. 73–80.

[97] A. N. M. Choudhury, K. C. Potter et S. G. Parker, “Interactive visualization for memory reference traces,” dans *Computer Graphics Forum*, vol. 27. Wiley Online Library, 2008, p. 815–822.

[98] B. Randell, “A Note on Storage Fragmentation and Program Segmentation,” *Commun. ACM*, vol. 12, n°. 7, p. 365, July 1969.

[99] J. E. Shore, “On the External Storage Fragmentation Produced by First-fit and Best-fit Allocation Strategies,” *Commun. ACM*, vol. 18, n°. 8, p. 433–440, août 1975.

[100] A. Bohra et E. Gabber, “Are Mallocs Free of Fragmentation ?” dans *USENIX Annual Technical Conference, FREENIX Track*, 2001, p. 105–117.

[101] S. Mamagkakis *et al.*, “Reducing memory fragmentation in network applications with dynamic memory allocators optimized for performance,” *Computer communications*, vol. 29, n°. 13, 2006.

[102] M. Hauswirth et T. M. Chilimbi, “Low-overhead memory leak detection using adaptive statistical profiling,” dans *Acm Sigplan Notices*, vol. 39, n°. 11. ACM, 2004, p. 156–164.

[103] W. De Pauw et G. Sevitsky, “Visualizing reference patterns for solving memory leaks in java,” dans *European Conference on Object-Oriented Programming*. Springer, 1999, p. 116–134.

[104] T. Tsai, K. Vaidyanathan et K. Gross, “Low-overhead run-time memory leak detection and recovery,” dans *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC’06)*. IEEE, 2006, p. 329–340.

[105] G. Carrozza *et al.*, “Memory leak analysis of mission-critical middleware,” *Journal of Systems and Software*, vol. 83, n°. 9, p. 1556–1567, 2010.

[106] N. Nethercote et J. Seward, “Valgrind : a framework for heavyweight dynamic binary instrumentation,” dans *ACM Sigplan notices*, vol. 42, n°. 6. ACM, 2007, p. 89–100.

[107] R. Matias *et al.*, “Measuring software aging effects through os kernel instrumentation,” dans *Software Aging and Rejuvenation (WoSAR), 2010 IEEE Second International Workshop on*. IEEE, 2010, p. 1–6.

[108] R. Matias, B. E. Costa et A. Macedo, “Monitoring memory-related software aging : An exploratory study,” dans *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*. IEEE, 2012, p. 247–252.

[109] V. V. Rubanov et E. A. Shatokhin, “Runtime verification of linux kernel modules based on call interception,” dans *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2011, p. 180–189.

[110] C. R. Attanasio *et al.*, “A comparative evaluation of parallel garbage collector implementations,” dans *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2001, p. 177–192.

[111] S. M. Blackburn, P. Cheng et K. S. McKinley, “Myths and realities : The performance impact of garbage collection,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, n°. 1, p. 25–36, 2004.

[112] L. T. Hansen, “Older-first garbage collection in practice,” dans *International Symposium on Memory Management (ISMM 2002*. New York, NY, USA : ACM, 2002, doi : 10.1145/773039.773042.

[113] P. Lengauer *et al.*, “A comprehensive java benchmark study on memory and garbage collection behavior of dacapo, dacapo scala, and specjvm2008,” dans *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ACM, 2017, p. 3–14.

[114] M. Hertz *et al.*, “Generating Object Lifetime Traces with Merlin,” *ACM Trans. Program. Lang. Syst.*, vol. 28, n°. 3, p. 476–516, mai 2006, doi : 10.1145/1133651.1133654.

[115] M. Hertz *et al.*, “Error-free Garbage Collection Traces : How to Cheat and Not Get Caught,” dans *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS ’02. New York, NY, USA : ACM, 2002, p. 140–151.

[116] P. Lengauer, V. Bitto et H. Mössenböck, “Accurate and efficient object tracing for java applications,” dans *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM, 2015, p. 51–62.

[117] N. P. Ricci, S. Z. Guyer et J. E. B. Moss, “Elephant tracks : Portable production of complete and precise gc traces,” *ACM Sigplan Notices*, vol. 48, n°. 11, p. 109–118, 2013.

[118] S. Fagan, “Tracing BSD system calls,” *DR DOBBS JOURNAL*, vol. 23, n°. 3, p. 38+, mar 1998.

[119] B. Gregg et J. Mauro, *DTrace : Dynamic Tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Prentice Hall Professional, 2011.

[120] F. C. Eigler et R. Hat, “Problem solving with systemtap,” dans *Proc. of the Ottawa Linux Symposium*, 2006, p. 261–268.

[121] The common trace format (ctf). [28 July 2018]. [En ligne]. Disponible : http://diamon.org/ctf/

[122] Babeltrace. [18 Jan 2019]. [En ligne]. Disponible : https://diamon.org/babeltrace/

[123] D. Toupin, “Using tracing to diagnose or monitor systems,” *IEEE software*, vol. 28, n°. 1, p. 87, 2011.

[124] A. Montplaisir *et al.*, “Efficient model to query and visualize the system states extracted from trace data,” dans *Runtime Verification*. Springer, 2013, p. 219–234.

[125] M. Jabbarifar, M. Dagenais et A. Shameli-Sendi, “Online incremental clock synchronization,” *Journal of Network and Systems Management*, vol. 23, nº. 4, p. 1034–1066, 2015.

[126] M. Seltzer, P. Chen et J. Ousterhout, “Disk scheduling revisited,” dans *Proceedings of the Winter 1990 USENIX Technical Conference*. Washington, DC, 1990, p. 313–323.

[127] M. Desnoyers, “Low-impact operating system tracing,” Thèse de doctorat, École Polytechnique de Montréal, 2009.

[128] B. Gregg. Radix tree. [18 September 2016]. [En ligne]. Disponible : https://lwn.net/Articles/175432/

[129] C. R. Lumb *et al.*, “Towards higher disk head utilization : extracting free bandwidth from busy disk drives,” dans *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*. USENIX Association, 2000, p. 7.

[130] B. L. Worthington, G. R. Ganger et Y. N. Patt, “Scheduling Algorithms for Modern Disk Drives,” dans *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS ’94. New York, NY, USA : ACM, 1994, p. 241–251, doi : 10.1145/183018.183045. [En ligne]. Disponible : http://doi.acm.org/10.1145/183018.183045

[131] Mpi parallel environment. [23 February 2016]. [En ligne]. Disponible : http://www.mcs.anl.gov/research/projects/perfvis/software/MPE/

[132] Jumpshot. [23 February 2016]. [En ligne]. Disponible : http://www.mcs.anl.gov/research/projects/perfvis/software/viewers/index.htm

[133] Tau performance system. [23 February 2016]. [En ligne]. Disponible : http://www.paratools.com/TAU

[134] B. Gregg. strace wow much syscall. [18 September 2016]. [En ligne]. Disponible : http://www.brendangregg.com/blog/2014-05-11/strace-wow-much-syscall.html

[135] M. Desnoyers et M. R. Dagenais, “Lockless multi-core high-throughput buffering scheme for kernel tracing,” *ACM SIGOPS Operating Systems Review*, vol. 46, nº. 3, p. 65–81, 2012.

[136] B. Gregg. Dtrace pid provider overhead. [18 September 2016]. [En ligne]. Disponible : http://dtrace.org/blogs/brendan/2011/02/18/dtrace-pid-provider-overhead/

[137] D. Domingo et W. Cohen. (2009) Red hat enterprise linux 5 systemtap beginners guide.

[138] R. Krishnakumar, “Kernel korner : kprobes-a kernel debugger,” *Linux Journal*, vol. 2005, nº. 133, p. 11, 2005.

[139] N. Ezzati-Jivan et M. R. Dagenais, “A stateful approach to generate synthetic events from kernel traces,” *Advances in Software Engineering*, vol. 2012, p. 6, 2012.

[140] H. Waly et B. Ktari, “A complete framework for kernel trace analysis,” dans *Electrical and Computer Engineering (CCECE), 2011 24th Canadian Conference on*. IEEE, 2011, p. 001 426–001 430.

[141] G. N. Matni et M. R. Dagenais, “Operating system level trace analysis for automated problem identification,” *Open Cybernetics & Systemics Journal*, vol. 4, p. 45–52, 2011.

[142] A. Montplaisir-Gonçalves *et al.*, “State history tree : an incremental disk-based data structure for very large interval data,” dans *Social Computing (SocialCom), 2013 International Conference on*. IEEE, 2013, p. 716–724.

[143] B. Gregg, *Systems Performance : Enterprise and the Cloud*. Pearson Education, 2013.

[144] Sysbench manual. [28 July 2018]. [En ligne]. Disponible : http://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf

[145] The ceph benchmarking tool. [18 Jan 2019]. [En ligne]. Disponible : https://github.com/ceph/cbt

[146] Rados bench. [18 Jan 2019]. [En ligne]. Disponible : http://docs.ceph.com/docs/master/man/8/rados/?highlight=bench

[147] Monitoring ceph with datadog. [18 Jan 2019]. [En ligne]. Disponible : https://access.redhat.com/documentation/en-us/red_hat_ceph_storage/3/pdf/monitoring_ceph_with_datadog/Red_Hat_Ceph_Storage-3-Monitoring_Ceph_with_Datadog-en-US.pdf

[148] Ceph storage monitoring zenpack. [18 Jan 2019]. [En ligne]. Disponible : https://www.zenoss.com/product/zenpacks/ceph

[149] R. V. Arumugam, L. Wujuan et S. H. W. Yi, “Scalable distributed meta-data management in parallel nfs,” dans *Advanced Applied Informatics (IIAI-AAI), 2016 5th IIAI International Congress on*. IEEE, 2016, p. 930–935.

[150] D. Hildebrand et P. Honeyman, “Exporting storage systems in a scalable manner with pnfs,” dans *null*. IEEE, 2005, p. 18–27.

[151] X. Zhang, S. Gaddam et A. Chronopoulos, “Ceph distributed file system benchmarks on an openstack cloud,” dans *Cloud Computing in Emerging Markets (CCEM), 2015 IEEE International Conference on*. IEEE, 2015, p. 113–120.

[152] M. Oh *et al.*, “Performance optimization for all flash scale-out storage,” dans *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*. IEEE, 2016, p. 316–325.

[153] N. Ezzati-Jivan et M. R. Dagenais, “Cube data model for multilevel statistics computation of live execution traces,” *Concurrency and Computation : Practice and Experience*, vol. 27, n°. 5, p. 1069–1091, 2015, doi : 10.1002/cpe.3272. [En ligne]. Disponible : https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3272

[154] N. Ezzati-Jivan et M. R. Dagenais, “Multi-scale navigation of large trace data : A survey,” *Concurrency and Computation : Practice and Experience*, vol. 29, n°. 10, p. e4068, 2017, doi : 10.1002/cpe.4068. [En ligne]. Disponible : https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4068

[155] A. Montplaisir *et al.*, “State history tree : an incremental disk-based data structure for very large interval data,” 2013.

[156] A. Montplaisir *et al.*, “Efficient model to query and visualize the system states extracted from trace data,” vol. 8174, p. 219–234, 2013. [En ligne]. Disponible : http://dx.doi.org/10.1007/978-3-642-40787-1_13

[157] H. Daoud et M. R. Dagenais, “Recovering disk storage metrics from low-level trace events,” *Software : Practice and Experience*, vol. 48, n°. 5, p. 1019–1041, 2018.

[158] A. M. Cheadle *et al.*, “Visualising dynamic memory allocators,” dans *Proceedings of the 5th international symposium on Memory management*. ACM, 2006, p. 115–125.

[159] N. Ezzati-Jivan et M. R. Dagenais, “Multiscale abstraction and visualization of large trace data : A survey,” *submitted to The VLDB Journal*, 2014.

[160] H. Daoud et M. R. Dagenais, “Recovering disk storage metrics from low-level trace events,” *Software : Practice and experience*, p. 1–14, 2017.

[161] C. Biancheri, N. Ezzati-Jivan et M. R. Dagenais, “Multilayer virtualized systems analysis with kernel tracing,” dans *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, Aug 2016, p. 1–6.

[162] C. Biancheri et M. R. Dagenais, “Fine-grained multilayer virtualized systems analysis,” *J. Cloud Comput.*, vol. 5, n°. 1, p. 69 :1–69 :14, déc. 2016.

[163] F. Wininger, N. Ezzati-Jivan et M. R. Dagenais, “A declarative framework for stateful analysis of execution traces,” *Software Quality Journal*, p. 1–29, 2016. [En ligne]. Disponible : http://dx.doi.org/10.1007/s11219-016-9311-0

[164] K. Kouame, N. Ezzati-Jivan et M. R. Dagenais, “A flexible data-driven approach for execution trace filtering,” dans *2015 IEEE International Congress on Big Data*, June 2015, p. 698–703.

[165] H. Daoud et M. R. Dagenais, “Multi-level diagnosis of performance problems in distributed systems,” *Journal of Systems and Software*, p. 1–14, 2017.

[166] T. Printezis et R. Jones, *GCspy : an adaptable heap visualisation framework*. ACM, 2002, vol. 37, n°. 11.

[167] M. Jurenz *et al.*, "Memory allocation tracing with vampirtrace," dans *International Conference on Computational Science*. Springer, 2007, p. 839–846.

[168] N. Nethercote et J. Seward, "Valgrind : A framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, n°. 6, p. 89–100, juin 2007.

[169] S. Cass, "The 2017 top programming languages," *IEEE Spectrum*, 2017.

[170] A. S. Tanenbaum et A. S. Woodhull, *Operating systems : design and implementation*. Prentice-Hall Englewood Cliffs, NJ, 1987, vol. 2.

[171] D. Goulet, "Unified kernel/user-space efficient linux tracing architecture," Thèse de doctorat, École Polytechnique de Montréal, 2012.

[172] D. Griswold, "The java hotspot virtual machine architecture," *Sun Microsystems Whitepaper*, 1998.

[173] Oracle hotspot. [28 July 2018]. [En ligne]. Disponible : https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html

[174] Oracle jrockit. [28 July 2018]. [En ligne]. Disponible : https://www.oracle.com/technetwork/middleware/jrockit/overview/index.html

[175] Eclipse openj9. [28 July 2018]. [En ligne]. Disponible : https://www.eclipse.org/openj9/

[176] Java language and virtual machine specifications. [28 July 2018]. [En ligne]. Disponible : https://docs.oracle.com/javase/specs/

[177] P. Akritidis, "Cling : A memory allocator to mitigate dangling pointers." dans *USENIX Security Symposium*, Washington DC, 2010, p. 177–192.

[178] J. Caballero *et al.*, "Undangle : early detection of dangling pointers in use-after-free and double-free vulnerabilities," dans *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, p. 133–143.

[179] B. Lee *et al.*, "Preventing use-after-free with dangling pointers nullification." dans *NDSS*, 2015.

[180] R. Hastings et B. Joyce, "Purify : Fast detection of memory leaks and access errors," dans *In Proc. of the Winter 1992 USENIX Conference*, 1991.

[181] R. Jones et R. Lins, *Garbage collection : algorithms for automatic dynamic memory management*. Wiley Chichester, 1996, vol. 208.

[182] F. L. Morris, "A time-and space-efficient garbage compaction algorithm," *Communications of the ACM*, vol. 21, n°. 8, p. 662–665, 1978.

[183] H.-J. Boehm, A. J. Demers et S. Shenker, "Mostly parallel garbage collection," *ACM SIGPLAN Notices*, vol. 26, n°. 6, p. 157–164, 1991.

[184] R. Jones, A. Hosking et E. Moss, *The garbage collection handbook : the art of automatic memory management*. Chapman and Hall/CRC, 2016.

[185] H. Lieberman et C. Hewitt, "A real-time garbage collector based on the lifetimes of objects," *Communications of the ACM*, vol. 26, n°. 6, p. 419–429, 1983.

[186] M. R. Jantz et P. A. Kulkarni, "Exploring single and multilevel jit compilation policy for modern machines 1," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, n°. 4, p. 22, 2013.

[187] M. Arnold *et al.*, "A survey of adaptive optimization in virtual machines," *Proceedings of the IEEE*, vol. 93, n°. 2, p. 449–466, 2005.

[188] D. E. Knuth, "An empirical study of fortran programs," *Software : Practice and experience*, vol. 1, n°. 2, p. 105–133, 1971.

[189] T. Kotzmann *et al.*, "Design of the java hotspot™ client compiler for java 6," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 5, n°. 1, p. 7, 2008.

[190] B. G. Sullins et M. Whipple, *JMX in Action*. Manning Publications Co., 2002, iSBN : 978-1930110564.

[191] Jvm tool interface. [28 July 2018]. [En ligne]. Disponible : https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html

[192] The java monitoring and management console. [28 July 2018]. [En ligne]. Disponible : http://openjdk.java.net/tools/svc/jconsole/

[193] Visualvm. [28 July 2018]. [En ligne]. Disponible : https://visualvm.github.io/

[194] Hprof : A heap/cpu profiling tool. [28 July 2018]. [En ligne]. Disponible : https://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html

[195] Honest profiler. [28 July 2018]. [En ligne]. Disponible : https://github.com/jvm-profiling-tools/honest-profiler/wiki

[196] The libunwind project. [28 July 2018]. [En ligne]. Disponible : http://www.nongnu.org/libunwind/

[197] M. Gebai et M. R. Dagenais, "Virtual machines cpu monitoring with kernel tracing," dans *Electrical and Computer Engineering (CCECE), 2014 IEEE 27th Canadian Conference on*. IEEE, 2014, p. 1–6.

[198] H. Nemati et M. R. Dagenais, "virtflow : Guest independent execution flow analysis across virtualized environments," *IEEE Transactions on Cloud Computing*, 2018.

[199] S. M. Blackburn *et al.*, “The dacapo benchmarks : Java benchmarking development and analysis,” dans *ACM Sigplan Notices*, vol. 41, n°. 10. ACM, 2006, p. 169–190.